UPTEC IT09 008
Examensarbete 30 hp
April 2009

Graphics System in Vehicle Electronics

Andreas Kjellgren

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Graphics System in Vehicle Electronics

Andreas Kjellgren

In this thesis three problems areas are studied related to embedded system and device driver programming: a GPS driver, the CAN Bus and study of graphics libraries suitable for embedded systems. The thesis has two parts: an academic study and an implementation phase based on the academic study. The Freescale i.MX31ADS development board together with ENEA's operating system OSE is used as a basis for the study and it is shown that OpenGL ES is best suited for the platform. Further the system can be complemented by the use of Mobile 3D Graphics, a Java based solution. A driver for the graphics port is implemented for Linux and OpenGL ES works using a graphics accelerator on the hardware. In the field of CAN communication an analysis of an existing driver is made. The driver has two shortcomings that lead to an incorrect priority order when multiple messages are sent simultaneously on the CAN bus. The main problem is that the bit, which tells if the data field of the CAN message fits in a single message, has the greatest impact on a CAN message priority. Another problem is that the signal numbers have not been assigned in a consistent manner. A design proposal and an implementation are made. The work with GPS is limited to theory and design in terms of creating the basis for the future creation of the driver. A survey of the interfaces that exist between the GPS module and other hardware is done and additional requirements from the rest of the system are highlighted.

Handledare: Detlef Scholle
Ämnesgranskare: Justin Pearson
Examinator: Anders Jansson
ISSN: 1401-5749, UPTEC IT09 008
Tryckt av: Reprocentralen ITC

Sammanfattning

Examensarbetet har genomförts i samarbete med ENEA Services AB i Kista som är ett konsultföretag med inriktning mot inbyggda system. Examensarbetet har utgjorts av tre problemområden, CAN kommunikation, en GPS mottagare och grafik för en hårdvara med medföljande skärm. Uppgifterna har gemensamt att de behandlar drivrutiner. En drivrutin är mjukvara (≈program) som har i uppgift att få någon eller några hårdvaruenheter att fungera tillsammans med ett operativsystem eller med andra datorprogram.

Ett grafikbibliotek är exempel på programvara som använder sig av drivrutiner för att fungera. En frågeställning i rapporten är vilka grafikbibliotek som passar att användas tillsammans med SHAPE. SHAPE är ett plattformsoberoende mjukvarulager som applikationer gör anrop igenom för att komma åt lägre mjukvarulager och hårdvaran. Under arbetet av frågeställningen kring lämpliga grafikbibliotek har hänsyn tagits till att det även ska passa operativsystemet OSE och hårdvaran, Freescale i.MX31ADS, som används i arbetet med grafik.

Ytterligare krav ställs på grafikbiblioteken och det grafikbibliotek som uppfyllde kraven och som passar bäst att använda med SHAPE är OpenGL ES. Mobile 3D Graphics uppfyller också kraven, men att använda Mobile 3D Graphics i stället för OpenGL ES skulle generellt kräva mer systemresurser. Dock har Mobile 3D Graphics utformats för att komplettera OpenGL ES snarare än att ersätta det. Javaanrop görs till Mobile 3D Graphics som i sin tur anropar OpenGL ES som hanterar utritningen av grafik.

En drivrutin för den port som den medföljande skärmen är kopplad till på hårdvaran har under arbetet blivit programmerad för operativsystemet Linux. Skärmen är kopplad till hårdvarans integrerade LCD kontroller, som i sin tur matas med information från hårdvarans minne. LCD kontrollern är en hårdvara som pratar med skärmens interface och som via en drivrutin initieras och konfigureras upp för att passa den inkopplade skärmen. Drivrutinen för LCD kontrollern används initialt när en ny skärm ska användas och när systemet startas upp, men behöver ofta inte göra något mer därefter. För att få applikationer och LCD kontrollern att läsa och skriva till samma del av hårdvarans minne används en framebuffer drivrutin. Framebuffer kallas den del av minnet som används till att mellanlagra grafisk data. Framebuffer drivrutin sköter allokeringen av hårdvarans RAM, sätter upp LCD kontrollern och informerar intresserade klienter (grafikbibliotek etc.) om var framebuffern finns.

I det andra problemområdet, CAN-kommunikation, har en befintlig drivrutin analyserats. CAN är en nätverksstandard som används framförallt inom fordonsindustrin. Den drivande frågeställningen i arbetet har varit att ta reda på vilka brister det fanns i den undersökta drivrutinen. Drivrutinen används i ett fungerande system. En undersökning av hur drivrutinen fungerar har lett till

att två huvudproblem hittats. Problemen ligger till grund för att meddelande skickas i en felaktig ordning när flera meddelande vill skickas på CAN-bussen samtidigt.

Det som avgör i vilken ordning CAN-meddelande skickas (meddelandets prioritet) är den binära representationen av CAN-protokollhuvudet på meddelandet. Det CAN-protokollhuvud som har lägst numeriskt värde har den högsta prioriteten och kommer att skickas först. De åtta första bitarna är indelade i två fält, datalängd och ett unikt signalnummer. De bitarna har granskats och sedan ändrats under den här studien. Det första problemet är att meddelande med lite data alltid skickas först, oavsett meddelandets relevans för systemet. Det andra problemet ligger i att signalnumren har tilldelats signalerna på ett sätt som gjort att de viktigaste signalerna inte har de lägsta numren.

En förändring av CAN-protokollhuvudet har genomförts som löser ovanstående två problem. De åtta första bitarna är nu istället indelade i tre fält; grupptillhörighet, datalängd och signalnummer. En analys har genomförts av alla signaler för att ta reda på hur ofta de används i systemet och hur viktiga de är för att systemet ska fungera. Sedan har signalerna blivit indelade i två grupper; systemsignaler och applikationssignaler. Systemsignaler skickas alltid före applikationssignaler. Den efterföljande biten avgör signalens datalängd och meddelande med lite data har fått behålla den högsta prioriteten. Det sista fältet, signalnumret, har blivit en bit kortare, men samtidigt strukturerats upp så att signalerna har fått en korrekt inbördes prioritetsordning.

Arbetet med GPS antennen har varit fokuserat på att teoretiskt och genom ett designförslag skapa en grund för att någon i framtiden ska kunna programmera en drivrutin för GPS antennen. Målet med drivrutinen är att GPS antennen ska skicka signaler till processorn och att processorn ska, efter en eventuell analys, skicka vidare informationen till andra delar av systemet.

Det huvudsakliga arbetet har legat i att dokumentera vilka krav GPS modulen ställer på det övriga systemet, exempelvis på meddelandestöd, överföringshastighet och på vilket kommunikationsinterface (sätt att prata) som används. GPS modulen i projektet ställer krav på att systemet ska kunna tolka NMEA-meddelande och ta emot information via ett UART-chip (kommunikationsinterface). GPS modulen kan skicka information i flera olika hastigheter, där någon av hastigheterna måste stödjas av systemet. Kraven från det övriga systemet på GPS modulen är också kartlagt, vilket påverkar utformningen av den framtida drivrutinen.

En av de teoretiska frågeställningarna som har behandlats är hur data kan plockas ut ur NMEA-dataströmmen som skickas från GPS modulen. En lösning är att programmera ett program som plockar ut innehållet mellan två kommatecken. Den lösningen är oberoende av om ett datafält är tomt eller innehåller många decimaler.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Problem Statement . . . 1
- 1.3 Method . . . 2
- 1.4 Limitations . . . 3
- 1.5 Contributions . . . 3

2 Theory **5**
- 2.1 Device drivers in OSE . . . 5
  - 2.1.1 Device drivers in OSE 5 . . . 6
  - 2.1.2 Device drivers in OSE Epsilon . . . 10
  - 2.1.3 Analysis of context regarding drivers . . . 10
- 2.2 Investigation of graphics libraries . . . 11
  - 2.2.1 Background . . . 11
  - 2.2.2 Choice of graphics library . . . 14
  - 2.2.3 OpenGL ES . . . 16
  - 2.2.4 Mobile 3D Graphics API for Java . . . 18
  - 2.2.5 EGL . . . 20
  - 2.2.6 Conclusions . . . 21
- 2.3 Evaluate existing CAN protocol header . . . 22
  - 2.3.1 Background . . . 23
  - 2.3.2 How it looks today . . . 24
  - 2.3.3 Problems in the implementation . . . 27
  - 2.3.4 Mapping of the signals . . . 27
  - 2.3.5 Design proposal . . . 27
  - 2.3.6 Conclusions . . . 28
- 2.4 Prerequisites for GPS antenna driver . . . 29
  - 2.4.1 What is GPS? . . . 29
  - 2.4.2 Conditions in this project . . . 31
  - 2.4.3 Conclusion . . . 39
- 2.5 Summary and Conclusions . . . 40

3 Design & Implementation **43**
3.1 Implementing graphics port driver 43
3.1.1 Used hardware 43
3.1.2 Connection of LCD 44
3.1.3 LCD Controller- and Framebuffer driver 45
3.2 Adjustments of CAN protocol header 48
3.2.1 Converting an OSE signal to a CAN message 49
3.2.2 Converting an OSE signal to a CAN message 51
3.3 Extend OSE 5 BSP with GPS support on i.MXS board . . . 51

4 Result **55**
4.1 Discussion and Conclusions 56
4.1.1 Graphics 56
4.1.2 CAN 58
4.1.3 GPS 59
4.2 Future Work 60
4.2.1 CAN 60
4.2.2 Graphics 60
4.2.3 GPS 61

A Abbreviations **67**

Chapter 1

Introduction

1.1 Background

Nowadays vehicles contain many embedded computers. One disadvantage is that the current use of such systems becomes static after the development process and the possibility to update the system is often limited to be done by professionals at service garages. Embedded systems are getting more important in the automotive industry and as the demands from customers increase, the lack of possibilities to modify the systems creates a problem.

The EU research project DySCAS[1] is an attempt to solve the problem. The project started in June 2006 and will after 30 months end during the period of this Master Thesis. The overall purpose of the project is to create a middleware, SHAPE[2], to make it easier to integrate consumer electronic devices in an automotive environment. ENEA AB are involved in the project through the creation of SHAPE and the develop of a demonstration platform to test the essential ideas which the project is built on.

The creation of SHAPE will benefit both consumers and vehicle manufacturers. Consumers will get a greater option to buy third party solutions and vehicle manufacturers will cut their expenses by not being forced to keep spare parts in stock for long term.

1.2 Problem Statement

The thesis will comprise one main task and two sub tasks. All three parts lies within the area of program or modifying drivers. The primary assignment of the thesis is to:

- Investigate which graphic library that best correspond to the demands from embedded systems like DySCAS and to program a prototype for

[1] Dynamically Self-Configuring Automotive System (DySCAS), http://www.dyscas.org/

[2] Self-configurable High Availability and Policy base platform for Embedded systems (SHAPE)

graphic drivers that are being used with the demonstration platform.

A natural first step in order to manage the task is to answer the question:
Q1. What graphics library is appropriate to use in SHAPE?

A further step would be to find out what choices that will be crucial for the system performance. The following question can contribute to that understanding:
Q2. What choices can be done to reduce needed system resources for graphics?

The sub tasks create an elementary understanding to cope with the main task. The first subtask is to:

- Modify the communication over the CAN-bus to work better on the demonstration platform

To be able to change the CAN communication, an analysis of the existing problem must be done first. To identify the problem, the following question must be answered:
Q3. What limitations/shortcomings imply communication through CAN in DySCAS?

The second less exhaustive tasks is to:

- Implement a GPS antenna driver on the demonstration platform

In order to justify the work to implement the driver and for the driver to provide any useful information to the rest of the system, the following questions are important:
Q4. What features can a GPS contribute to DySCAS?
Q5. How do you pick out data from the GPS stream?

The questions *Q1-Q5* will be addressed and answered in the theory chapter of the report.

1.3 Method

The approach for this master thesis will be to start with an academic literature study based on books, technical manuals and research papers within the area of CAN, GPS and several graphic libraries. The development process and design of drivers will have a predominant part of this thesis work. The thesis report will contain the results of the literature study and the results from the design and implementation part of the thesis.

The development will be carried out at an existing setup consisting of, among others, a Freescale development board with a I.MX31 processor

and two telematic units from an automotive environment communicating through CAN.

The purpose of making an analysis of the GPS system, the CAN-bus and the graphic libraries are to get a better understanding of the possibilities and limitations of such technologies. A basic understanding of GPS is also needed to pick out information from the protocol to be able to pass on that information to other parts of the system.

1.4 Limitations

The work will conclude a demonstration setup to show the results of my work. The three tasks are all using different hardware, so a demonstration will be implemented for the various components individually. Since the thesis work is only 20 weeks long, that parameter will determine how advanced the graphical presentation will be. With more time the different parts of the thesis can be linked together to show that they work jointly.

1.5 Contributions

A brief summary of a framework to write drivers for the operating system OSE is given. There are essentially two ways to write drivers that are recommended, signal-based interface and system call interface.

Based on some requirements an analysis has been made to find out which the appropriate graphics libraries are to use in embedded systems. A focus in the search has been made to suit the hardware i.MX31ADS and ENEAs operating system OSE 5. One of the requirements has been that the graphics libraries must be operating system independent as a future goal is to add a graphic HMI (Human-Machine Interface) to the middleware SHAPE. SHAPE is at the moment implemented in combination with OSE and Linux. A working driver to be able to present graphics on a display attached to the hardware i.MX31ADS has been developed based on an existing BSP (Board Support Package) for Linux. The driver is based on that a storage area, the framebuffer, is used from where the LCD controller (hardware) retrieves data which is then presented on the display. It is all made possible by controlling software that allocates memory for the framebuffer, informs interested parts of the system (e.g. graphics libraries) about the framebuffer, configure the LCD controller on the basis of size and address of the framebuffer and specific settings for display (e.g. resolution, color information).

An investigation is made on the CAN communication in a working system. Two main problems are identified and corrected, which otherwise contributes to an incorrect order when the messages are sent. The order in which CAN messages are sent is determined by the construction of the CAN protocol header. The length of data field in the CAN message has

the biggest impact on the order in which the message is sent and a unique signal number has been assigned to the signals in a non-structured way. A modification of the CAN protocol header has been introduced and the signal numbers are assigned based on how often the messages are sent and how important they are for the system to work.

A study of hardware and conditions to complement an existing BSP (for OSE 5) with a device driver for a GPS antenna has been made. The hardware is custom-built and aimed for use in a vehicle. The focus has been on the theoretical to provide a basis for how the device driver should be designed. A design proposal will also be provided on a rough construction of the driver. An analysis of the GPS receivers output has been made and a design proposal of a way to pick out information from the stream of data is given.

Chapter 2

Theory

The theory chapter is divided into four parts. It begins with a description of the ways to write device drivers for the operating system OSE. That knowledge is necessary to sort out several of the tasks this thesis contains.

This is followed by a study of graphics libraries suitable for embedded systems. An adaption was made to especially fit the hardware, i.MX31ADS, used in the thesis graphics work. Based on the requirements of the report two graphics libraries, OpenGL ES and Mobile 3D Graphics API for Java, remain which could be combined to obtain a greater width of the range of applications. Should only one of these graphics library be used the choice is OpenGL ES, as it is adapted for applications written in low level languages.

An evaluation is then performed of an existing CAN communication and the way in which signals are given priority when there are several signals that want to be sent simultaneously. Problems with the current solution are identified and a solution to the problems are given, which in a later section of the report also will be implemented.

A description is made about how GPS works and what characteristics and conditions that this projects GPS has. The GPS will be used with an existing system and therefore a description of the adjustments that need to be made to the GPS to function in this system is made.

Finally, a summary of all the sections in the theory chapter will be provided. The questions addressed in the introduction of the report will be briefly answered.

2.1 Device drivers in OSE

This chapter explains the fundamentals of implementing device drivers in OSE 5 and OSE Epsilon. Implementing device drivers is a large part of this thesis and to follow given guidelines will facilitate the work. There is nothing in OSE that forces you to use device drivers. In some situation, it might be easier to let the application deal directly with the hardware.

This is however not the case in this thesis. The following sections is based on information from OSE Core User's Guide[7], OSE Device Drivers User's Guide[8] and C166 Board Support Package User's Manual[6].

2.1.1 Device drivers in OSE 5

As a support for writing device drivers for OSE 5, ENEA supplies a framework called "device driver architecture" (DDA). When implementing device drivers according to this framework you are recommended to follow some basic rules:

1. Each device driver should only deal directly with a single device. Services from other devices should go through the device resource management (DRM) interface (see 2.1.1).
2. Device drivers shall use the DRM interface for management of board resources and configuration parameters. Every driver provides one function called DRM entry, which is the entry point for all the DRM functionality.
3. When relevant, device drivers must be able to be instantiated multiple times to handle multiple instances of similar devices.

The DDA framework is composed of the following four entities (See Figure 2.1):

- Device manager
- The device resource management interface
- Device driver architecture utilities library
- Device drivers

Device manager

One could say that the device manager has the central role of the DDA. The device manager's main task is to act as a negotiator when a process wants to access functionality from a device driver. It is the device manager that looks up the suitable device driver with the corresponding features that the process asks for.

The device manager defines a signal interface, which is used for adding new device drivers to the database. Every device driver reports their services to the device manager and the device manager keeps a database of the installed device drivers. When the device manager passes configuration information to device drivers it also constructs a device path for the driver. The device manager uses this information for building a logical device tree,

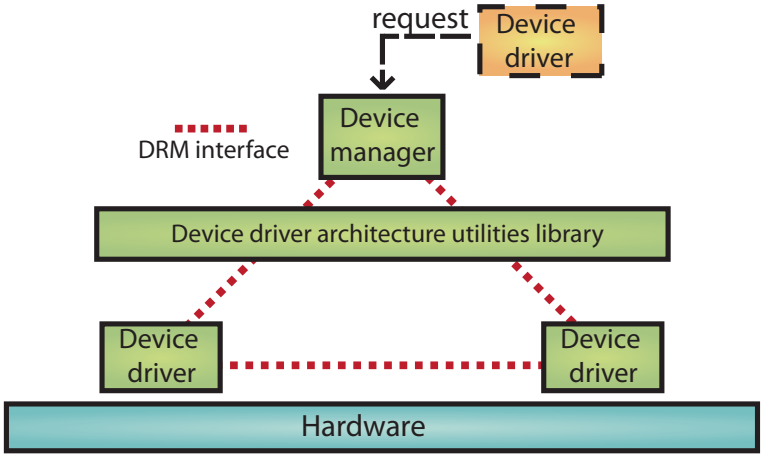

Figure 2.1: Device driver architecture overview

which models the way hardware depends on each other. The device manager is the root of this tree. When a new device is found the device manager searches the database to find a match for the device.

Device resource management interface

Some drivers provide services to other device drivers. This is done via the DRM interface. A DRM call is a function call that is issued from one device object to another. It is through the DRM interface device drivers announce their functionality, during the system start phase, to the device manager. This service is used to allocate resources from other drivers, often the parent in the logical device tree, but also resources from the device manager.

Device driver architecture utilities library

The driver architecture utilities library contain wrappers for DRM calls and commonly used functions that device drivers can use.

Device drivers

A device driver consists of a collection of functions used to implement a physical or virtual device functionality for the benefit of a higher level device client. Technically the device driver is just a collection of operating system independent functions used to handle the device hardware. Device drivers provide a service to applications via a device client interface. DDA provides two different interfaces for device clients (often applications) to communicate with the device drivers. The two interfaces are presented in the following sections.

Signal based interface

The first category of device drivers are implemented using a signal based interface. These drivers usually execute, in relation with interrupt processes, by executing code in the device client. Through this interface, the device client (User mode privileges) and the device driver (Supervisor mode privileges) execute in different process contexts. The advantage of this way of communication is that there is no need to worry about mutual exclusion to shared resources. This is however, seen from another perspective, also a drawback since you have to transfer data between different memory buffers. The signal based intergace is the interface that makes it easiest to create a safe system, since you never let the device client to execute in supervisor mode. Another drawback is that there is a limitation in the size permitted data per signal.

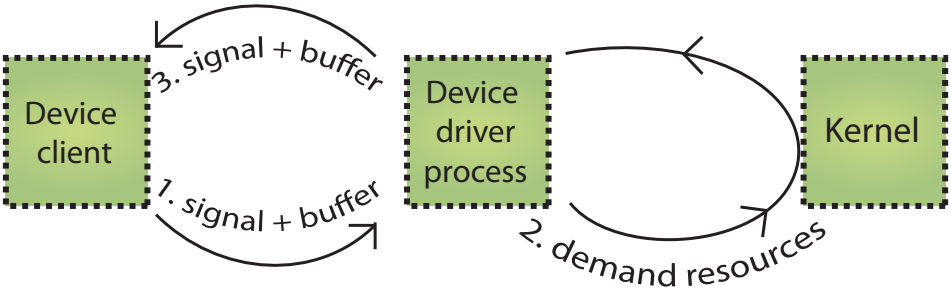

Figure 2.2: Signal based interface

This can be seen as a device client sends a request (1) and hands over a buffer to the device driver process. The client device may send several requests without that any data is lost. When the buffer(s) has been read by the device driver process, the device process demands the asked hardware resources (2) from the kernel and returns a reply (3). If the device driver have data available, but no buffer to write to, the device driver is forced to throw the data away. This means that a device driver never can overflow a device client with data. See Figure 2.2.

System call interface

The second category is made up of communication between device clients and device drivers in the form of system calls, BIOS[1]. The method of using system calls allows device clients to call the executive, or other components that have registered with the BIOS, without knowing the address at which the executive or the component is located. The BIOS architecture is built

[1]The OSE BIOS component has nothing to do with the Basic Input/Output System (BIOS) term in the desktop PC world

with a table. Each table element consists of a string containing the driver name and a pointer to the entry function of the driver. The handle (see the "Open" call defined below) represents the table index for a corresponding driver name. To use the BIOS system call is also a way to change privileges level for user mode code to execute in supervisor mode in a controlled manner.

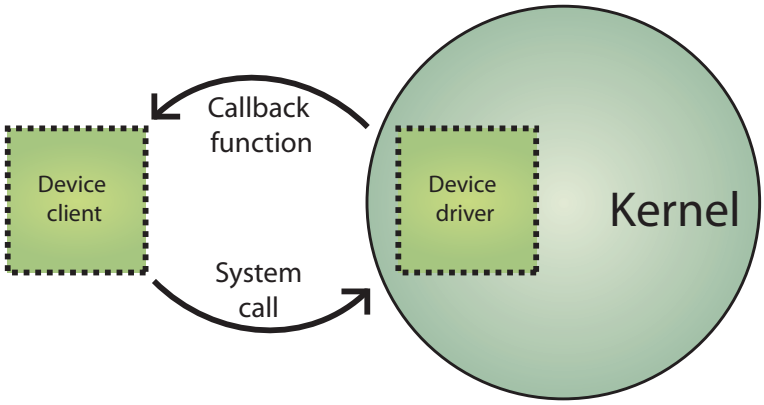

Figure 2.3: System call interface

Once the device client has established a communication to the device driver through a system call, most of the interaction with it will be through a few calls to the device driver and a few callback functions in which the device driver calls the device client (See Figure 2.3). The system call interface provides four important operations:

Init - This call initializes the BIOS module and clears the table content.

Install - A device driver can perform an install operation by providing a name and entry point pair. This is the way for the device driver to tell the device manager about its services. After registration of the device driver via the Install command, the driver is accessible for the client device. The information is stored in an internal table by BIOS.

Open - This call is used by the device client to find the registered entry points. If the right entry point is found, this is returned as a handle.

Call - To execute functions through the given handle, the device client can call that entry point. BIOS will then jump to the registered entry point and enable execution.

This is the method the OSE Kernel uses internally to provide all the system calls to the applications that are running in user mode and/or are not linked to the core module.

An advantage of this method is that there is no need to shuffle data, since the device client is able to call functions exported by the device driver through a supplied handle, which implies faster execution. A major drawback is the problem if a device client with corrupted data gets access to execute in supervisor mode. This enhances the need for well written functions that gets exported from the device driver and the risks for loopholes are larger.

2.1.2 Device drivers in OSE Epsilon

OSE Epsilon is a small and low-cost real-time operating system optimized for deeply embedded microcontroller applications. For OSE Epsilon, there is no framework on how device drivers could be written. The interface that is described earlier, with communication via system calls (BIOS) and callback functions, also works in the OSE Epsilon. It is also possible to implement the communications directly from the application to the hardware. Calls directly to the hardware can in some cases increase the speed, but it provides less structure, especially in larger systems.

2.1.3 Analysis of context regarding drivers

CAN driver

Controller Area Network, abbreviated CAN, is a networking class and a network standard developed for the automotive industry. The usage in this project is to allow several nodes in the system to communicate and send messages to each other. The CAN drivers are going to be implemented to the operating system OSE Epsilon. Even though the signal based interface is not a choice, the system call interface would have been chosen anyhow. This is because CAN is used as a data communication protocol and has to send information fast, to not slow down the entire system. At the same time, since data will be transmitted, there is waste of resources to constantly copy data between buffers.

Graphics port driver

Both the device driver for the GPS antenna and the driver for the graphics port are intended for OSE 5. Almost the same arguments can be used, as for CAN, to motivate why the graphics port driver should be implemented using the system call interface. Graphics claims many resources and send large amounts of data, so there is important that the communication is rapid between the application and the driver. As it is mentioned earlier, there is a limit in the signal size that is used within the signal based interface, and this might limit the capacity of the system.

GPS anthenna driver

The driver for the GPS antenna does not have the same requirement on fast information exchange. Furthermore, it is not much data to be sent at a time, so it is not a bad choice to implement these drivers using the signal based interface. This implies an easier way to make the system safe, since the device clients never gets to execute in supervisor mode.

2.2 Investigation of graphics libraries

In the following sections, a survey is done about how graphics system has emerged and what factors that has contributed to the development. The main purpose of the study is to identify what graphics libraries that meets the requirements of embedded systems, which are operating system independent, is able to create both 2D and 3D graphics and also is an open standard. The outcome was only two graphic libraries, it may be because of that graphics for embedded systems have not been so long in comparison with the graphics of desktop computers.

The two graphics libraries that meet the requirements are OpenGL ES and Mobile 3D Graphics. OpenGL ES can be used either by itself or it can be supplemented with the Mobile 3D Graphics to extend the functionality to also cope with Java. The choice may be made depending on needs. For the hardware that the investigation is tailored to the complement would be interesting to have, since other parts of the system make use of Java. To use only Mobile 3D Graphics is not recommended in an embedded system because of the overhead that Java code provides in relation to a low level programming language like C.

A description of EGL is also made. EGL is an interface between rendering APIs such as OpenGL ES and the underlying native platform window system. It is optional to use any EGL, but it makes it easier for the developer.

2.2.1 Background

In early 1980's mostly 2D-graphics standards and hardware-oriented industry standards existed, at the same time as hardware traders created specialized graphics workstations and introduced graphics libraries for their own hardware. The beginning of 3D graphics computing was characterized by very expensive and highly primitive hardware and software.

Beginning around 1990 3D standards like OpenGL appeared on the market. These standards were developed and showed more and more parts of a graphics rendering[2] pipeline. Still the workstation hardware was very

[2]Rendering is the process of generating an image from a 3D model by means of a software program[5]

sophisticated and expensive[18]. Expressions like ubiquitous computing and interactive mobile graphics was unimaginable. Mobile devices, such as cell phones, used small monochromatic displays, slow CPUs[3] and little memory[4].

The resources have improved though. The CPU development has led to smaller and higher performance CPUs and the amount of available memory has increased. High-performance 3D graphics applications have moved during the last decades from large sized systems with specialized hardware and software to regular low-cost PC-hardware and gaming devices[18, 22]. This is the result of an agreement of common standards like, for example, OpenGL (manufacturers want open standard solutions in order to reduce development costs and to shorten product development times) in combination with the rapidly improving graphics technology.

There has been a fast progress within mobile graphics technology the last five years, and it has followed a similar path as on PCs: early copyright protected software engines running on integer hardware paved the way to open source standards for graphics hardware acceleration[22]. In a latter phase of the technical evolution, mobile computing devices have been used in other areas than mobile telephony and PDA[4] [18].

How does it look today?

Now everybody demands for high-performance user interfaces on his everyday-computing environment. Perhaps the most important enabler to the widespread use of mobile devices has been the tremendous advances in display technologies; it is now common with 24-bit displays with VGA[5] resolution, according to the article "The State of Art in the Mobile Research"[4]. That development was first fueled by the demand from digital cameras, though now the greatest demand probably comes from mobile phones[22].

This opened up excellent opportunities for all user-interaction based applications as it allowed importing more realism and understanding into human-to-process communication information. Automotive engineering introduces several user interfaces and entertainment systems for next generation cars. Although all industrial sectors ask for high performance graphics, they tend to focus on different aspects due to the still existing limitations of their graphics devices. Embedded software is very dependent on the hardware of a product. Features of the software are used in different ways between the products, because of differences in their specifications[3]. This leads to a demand to optimize the graphics standards in order to make them suitable for special embedded and small-scale devices [18].

[3] Central Processing Units
[4] Personal Digital Assistance
[5] Video Graphics Array

According to Antochi et al.[16] the amount of users of interactive 3D graphics applications is predicted to increase significantly in the near future. The usage of 3D graphics in embedded and small scale systems are growing substantially in all areas of life. Especially popular within the area of gaming applications, 3D graphics are also distributed on a wide range of devices, such as mobile phones, car radios, navigation systems, PDA and devices with 3D-style user interfaces, public and private infotainment systems and interactive entertainment[10]. The application that drives the graphics technology development on mobile devices most seems to be the same one as on PCs, namely gaming[22].

Until a few years ago there was no commonly accepted API (Application Programming Interface) for 3D graphics on mobile devices[5]. Recently, as a result of the development within graphics technology and the high interest in embedded 3D graphics, APIs accommodated for mobile 3D graphics such as OpenGL ES and Java Mobile 3D Graphics API for J2ME have appeared. A 3D graphics API is a prerequisite for easily developing 3D applications[17].

The beginning of 3D graphics in embedded systems

The progress of mobile devices has reached a point where interactive 3D graphics are possible to initiate and use. 3D graphics on mobile devices was initially introduced on the market by the Japanese operator JPhone during 2001. JPhone is the company behind HI Corps Mascot Capsule engine (that is used by Ericsson). After a while other Japanese operators started to use the same engine. The functionality in the first versions of Mascot Capsule was limited, but it was later extended with a richer feature set and a lower lever API. Outside of Asia no copyright protected 3D engine became an actual standard, since the leading telecom companies used several different 3D engines [21].

The first standard real-time 3D graphics API for mobile devices, OpenGL ES, was published in July 2003[20]. OpenGL ES is a subset of the long time established graphics library OpenGL for desktop computers. OpenGL ES is a low level rasterization interface available from native C(++) code.

An API built on Java, JSR-184 (later on referred to as Mobile 3D Graphics), was introduced shortly after OpenGL ES. This API supplies higher level functionalities and also a binary scene graph file format called M3G. A M3G graph file can easily be parsed in and rendered. Mobile 3D Graphics is designed to use OpenGL ES for rasterization and the rendering pipeline is hidden for a programmer using the J2ME environment, whereas the rendering pipeline features of OpenGL ES are typically visible to a native C(++) programmer[20].

The first software implementation of OpenGL ES and Mobile 3D Graphics came from Nokia, Hybrid Graphics, HI Corporation and Superscape. In the end of 2004 there was also several hardware designs intended for these

APIs. The two interfaces are designed to complement each other and can take advantage of the same underlying rendering engine [21].

2.2.2 Choice of graphics library

There has been a continuous development of graphics APIs in the last decade. APIs that have not been able to add new features or functionality have disappeared. Extensibility is especially important for mobile APIs as the products evolve faster than the desktop ones.

Simplicity to adopt an API is important, since a "programming interface" is no good if no one implements it, or if developers choose not to use it. This can happen if the API is inappropriate for the environment, does not have support for development tools, is hindered by patents or not promoted enough. Open standards are essential to avoid such barriers[21].

The choice of graphics library has been made with the following software requirements (SWRQ):

SWRQ 1 - It shall be suited for embedded systems

SWRQ 2 - Operating system independent

SWRQ 3 - It should be able to create both 2D and 3D graphics

SWRQ 4 - It shall be an open standard

Since the implementation will be carried out at an embedded system, there is an obvious need that the graphics library is suited for this purpose. The main objectives are to find a solution that does not use much memory and processing power.

The graphic library shall not be bound to a specific operating system (except possibly ENEAs operating system OSE). This will rule out the accomplishment of this thesis work.

Many user interfaces and background images consist of 2D graphics. This is of great use, but it is even more important, for the future use, to support the implementation of 3D graphics. To be able to combine 2D and 3D graphics is also a requirement.

Open Standards are available for all to read. They are free for all to implement, with no royalty or fee, but certification of agreement by the standards organization may involve a fee. Open standards also create a fair and competitive market for the implementation of the standards. The organizations that administer the standard do not favor one implementator over another.

With those requirements there were not many choices left. There were basically only two choices left, which in fact complement each other rather than compete.

OpenGL ES and Mobile 3D Graphics are the two major platform independent graphics APIs. Mobile 3D Graphics actually works as a layer between Java and OpenGL ES (you make Mobile 3D Graphics calls from Java, and Mobile 3D Graphics invokes, in turn, OpenGL ES to manage the graphics rendering).

Using OpenGL ES is the main solution to the problem, but the possibility of complementing the graphics library with Mobile 3D Graphics exists. Mobile 3D Graphics does not necessarily have to use the OpenGL ES, but that stand alone solution would not be equally economic with the resources, since it is based on a high level 3D API.

OpenGL ES also includes a specification of a common platform interface layer, called EGL. This layer is platform independent and may optionally be included as part of OpenGL ES, but the developer may choose to define their own platform-specific embedding layer. OpenGL ES, Mobile 3D Graphics and EGL will be presented in more detail in later sections.

Why not use OpenGL?

Graphical HMIs (Human-Machine Interfaces) are gradually making more use of the OpenGL rendering API as a standard for rendering graphics to displays. At the same time software vendors have brought in OpenGL in different ways as the rendering API they support. This tendency is supported by the increasing call for hardware accelerated graphics subsystems and commercially available driver software. However, for systems designed for embedded environments, full OpenGL is not sufficiently optimized for being a standard. The full OpenGL library contains too many features that are not applicable for systems with restricted resources like embedded systems. In order to reduce driver complexity OpenGL subsets must be defined[25].

OpenGL is usually implemented through a driver architecture. OpenGL drivers comprise a low-level interface to the rendering hardware, while it features a high-level interface to applications that will request hardware resources. OpenGL is a standardized API that has evolved over many years. It has achieved an extensive adoption in the simulation, gaming, CAD[6], and professional graphics markets.

During the evolution of OpenGL the size of its drivers have grown. The driver implementation is too large for embedded hardware and the resources of the hardware are lacking. An example is that calculations in OpenGL depends to a great extent on floating point arithmetic, but the support for floating point arithmetic is often limited or even missing on mobile devices[18]. Nowadays OpenGL drivers, for a modern desktop graphics card, can without doubt run into several hundred of thousands of lines of code. Embedded versions of OpenGL can consist of less code depending on what subset of OpenGL it is[25].

[6] computer-aided design

Combining two graphic libraries

For graphics on embedded systems OpenGL ES, with the possibility to add Mobile 3D Graphics on top, enables hardware accelerated or software-based interactive 3D applications with small demands on the memory and computation resources[23]. OpenGL ES is as a matter of fact the standard for embedded applications. It is a main standard in the embedded avionics market due to its power and cross platform nature[25].

2.2.3 OpenGL ES

The demands and interest for using 3D graphics on mobile devices have increased successive during the last decade. By using 3D technologies, the mobile devices gain access to powerful tools. To provide such a graphic library for the ubiquitous computing market, performance and memory limitations are some of the challenges to master[26].

When designing an open low-level multiplatform graphics API[7], OpenGL constitutes an excellent benchmark. OpenGL has long been a well-used and accepted API for graphics rendering on desktop computers. OpenGL is the main 3D API for MacOS, Linux, most Unixes and it is available for Windows. It is itself based on previous APIs and has clear design principles that many 3D programmers are used to. OpenGL was not designed to minimize the size of the API and its memory footprint (size of implementation). During its lifetime it has grown bigger in form of complexity and the increasing number of functions along with the adaption to new hardware platforms [21].

Since OpenGL is too big and is designed for more powerful hardware than embedded systems generally have, the graphic library needed to be cleaned of redundant functionality and rarely used functions to better apply to the requirements from mobile devices. The majority of the work was put on minimizing the need of resources that are insufficient on mobile devices. The Khronos Group [14], a member-funded industry consortium that is focused on the creation of open standard and royalty free APIs for handheld and embedded devices, took the responsibility to implement the changes and create OpenGL ES, where ES is an abbreviation of embedded systems.

The evolution of OpenGL ES

Version 1.0 of OpenGL ES was designed with OpenGL 1.3 as a starting point and it had as objective to form an API that enabled an efficient implementation on mobile devices whose hardware only supported integer arithmetic. To deal with the compactness issue all alternative ways to end up with the same result was reviewed. The functionality that was thought to have the

[7] An API provides a defined method for developing applications software.

widest scope of use was kept, while the other functions were striped away. Since some features are more basic than other, hard to emulate using the remaining functions and/or the API can not do without these there became small overlaps. Among other things all primitives was simplified. Only point, line and triangles were kept, whereas quadrangles and common polygons were removed. These are typically presented by numerous triangles in this reduced API.

The second revision, OpenGL ES 1.1, was released during 2004. In contrast to the first version, that was created as minimal as possible, OpenGL ES 1.1 aimed at systems with better resources. Five optional and two mandatory features were introduced. The largest update within the area of visual quality was a further elaborated function and a better support for texture mapping.

OpenGL ES 2.0 is currently the latest version of the API for embedded systems. It is defined relative to the OpenGL ES 1.1 and OpenGL 2.0 specifications and introduces a programmable 3D graphics pipeline with the ability to create shader and program objects using the OpenGL ES Shading Language [2]. OpenGL ES 2.0 is not backward compatible with OpenGL ES 1.x[8], since it does not support the fixed function pipeline that OpenGL ES 1.x applies. Thus the OpenGL ES 2.0 programmable vertex shader replaces the fixed function vertex units implemented in OpenGL ES 1.x. Although OpenGL ES 2.0 is not backward compatible with 1.x, the APIs are designed so that the drivers from both versions can be implemented on the same hardware, allowing 2.0 and 1.x applications to be executed on the same device [22].

Benefits when creating an API based on another API

Making the embedded API similar to an already well-known graphic library has several advantages [21]:

- There already exists lots of useful literature that eases the adoption of the API among developers.
- The standardization process can evolve smoother and faster.
- The knowledge of strong and weaker points in the original API makes it easier to accept similar weaknesses in the upcoming API.

OpenGL ES has now been adopted as the main 3D API for Symbian OS, an open mobile operating system mainly for cell phones and PDAs.

[8] Here 1.x is either 1.0 or 1.1

2.2.4 Mobile 3D Graphics API for Java

The Mobile 3D Graphics (JSR 184) is an optional high-level 3D API for mobile Java that facilitates the manipulation of 3D graphics elements on mobile devices. The Mobile 3D Graphic API 1.0 is fully specified in JSR 184 [13]. Mobile 3D Graphics uses the same rendering model as OpenGL ES. The two standards were developed concurrently, partly by the same developers. It would be possible that if a single device supported both OpenGL ES and Mobile 3D Graphics and the same essential graphics engine, particularly hardware accelerated, it could take advantage of both APIs. In fact, it is even recommended to implement the Mobile 3D Graphics as a complement to OpenGL ES [22].

The need for Java

OpenGL ES is a low level API intended for applications written in C. Mobile devices have for a long time been closed platforms in the sense that no new applications can be installed, by the end user, after the device is purchased. On the other hand mobile devices are opening up. Although there are some operating systems (e.g. Symbian OS and embedded Linux) accommodated for embedded systems that allow end users to install their native applications written in C or even assembler, that number is small compared to the devices that support installation of Java applications. The mobile Java 2 Platform, Micro Edition (J2ME) allows the same applications to run on many different devices and operating systems. Consequently there is practical to support a 3D API for J2ME as well [22]. Again, it was the Khronos Group that took the responsibility to introduce such a solution.

The idea to elaborate a subset of an existing high level API so that it agrees to the J2ME environment and to make the subset more compact to work with embedded systems was attracting in the same way as for OpenGL ES. A natural choice of a desktop API to use as a starting point to create the upcoming Mobile 3D Graphics would be Java3D. Java3D is a high level API that gives extended support for scene graph operations and a certain support for animations and compressed geometry. Java3D is based on earlier scene graph frameworks, like Open Inventor and VRML, and can be implemented on top of OpenGL. Therefore a subset of Java3D could probably be designed and implemented on top of OpenGL ES, so that both APIs could take advantage of the same underlying rendering engine.

However, it was not possible to simplify Java3D in a satisfactory way to end up with a subset. The preliminary work to clear all redundant functionalities from Java3D showed that too large modifications was needed and that several changes and extensions would have been forced, which in the end would have resulted in a completely new API. The decision was taken to develop and design a completely new API instead, which assumed and bor-

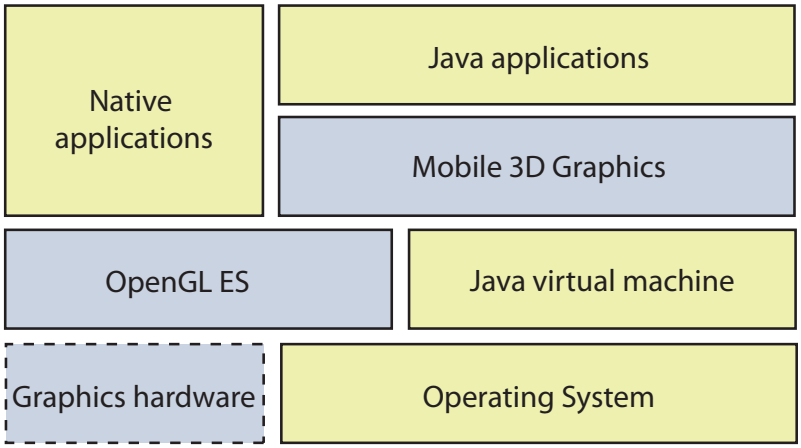

Figure 2.4: Mobile 3D Graphics can work as a layer between Java and OpenGL ES. You make Mobile 3D Graphics calls from Java and Mobile 3D Graphics invokes OpenGL ES to manage the graphics rendering.

rowed many basic design rules from Java3D. The new design largely offers the same fundamental functionality as Java3D, but avoid unnecessary and overly generalized class hierarchies that prevented the simplification of Java 3D. This resulted in that the footprint (number of classes and methods) of Mobile 3D Graphic got considerable smaller than that of Java3D[21, 22].

Goals with Mobile 3D Graphics

When designing Mobile 3D Graphics the major goal was to overcome the performance penalty caused by Java virtual machines[23]. The main reason for choosing a high level API over a low level one is that Java code is generally slower, up to 10-20 times, than well written native code, especially on mobile devices. This depends on the virtual machine execution model and the built in protection against illegal memory accesses and memory leaks [21]. The overcome of the performance penalty was fulfilled by raising the abstraction level of the API over that of OpenGL ES, allowing the applications to reduce the amount of Java code required for regular animation and rendering tasks, see Figure 2.4

Applications needs to do numerous other tasks in excess of rendering, for example object database or scene graph traversal and object and camera transformations. If the 3D rendering is a comparatively small part of the total workload, the gain of hardware accelerating the rendering part will not considerable accelerate the full application. This suggest that if a high level API includes lots of additional tasks, it is better that processing intensive features is provided by adding them into the graphics API and implement them in native code, only leaving the control logic to Java [22].

In the same way as with the graphics engine, the applications and programs using the engine needs to be small. Since Java applications often can be installed, especially on mobile phones, over a wireless connection, and many users have high demands on the download times, the size of the application is very important. To provide high level functionality such as scene graph and animation capacity in the graphics API together with the competence to encode geometry, textures and animation data into small binary files, allows the final applications to be more compact or include more content. It is essential that the low and high level APIs complement rather than compete with each other and support similar hardware models so that the same hardware can accelerate both of them [21].

Combining 2D and 3D graphics

Java offers a comprehensive set up 2D graphics rendering features. Despite the evolution that has brought in more 3D graphics on embedded systems, 2D graphics is still used a lot for user interface elements and background images. Mobile 3D Graphics API allows programmers to take advantage of the existing functionality. The main concern when mixing 2D and 3D rendering is the use of two different renderers that needs to be synchronized. A large number of rendering target types might be supported and several combinations of where the renderers are implemented can exist. For example, one renderer can be accelerated, while the other is implemented in the software.

Mobile 3D Graphics handles this by making these problems explicit. All 3D rendering has to begin and end with specific method calls. These method calls tells the other rendering processes that it has the ownership of the target, and only at these points the synchronization between 2D and 3D graphics occurs, which results in that drawing of 3D graphics while target is bound to 2D rendering is undefined. A rendering target can be bound and released as many times as wanted, but with a potential drop in performance by every such synchronization. The API itself does not contain any information about what rendering targets it support, but instead accepts a generic Java object that is checked for compatibility during the implementation. By this approach it gets easier to accept new rendering target types in the future [21].

2.2.5 EGL

EGL[9] is a library specification [18] and a platform independent API originally designed to facilitate the integration of OpenGL ES into the native windowing system of the operating system. It is similar to GLX [1] (OpenGL Extension to the X Window System) and allows interfacing and control of

[9] Embedded-System Graphics Library

resources such as frame buffers, memory, window and platform resources. There is no requirement to provide EGL when implementing OpenGL ES. On the other hand the OpenGL ES API does not state how to create a rendering context or how the rendering context gets attached to the native windowing system [23].

To operate correctly the commands in OpenGL ES is dependent of EGL before any rendering can begin. EGL is needed to create a rendering context, which stores information about the proper OpenGL ES state. The rendering context is actually also where all of the other pieces of the graphics library come together to in fact display bits on an output device or display. The rendering context needs to be attached to an appropriate surface before the rendering can begin.

The drawing surface defines the required buffers needed for rendering e.g. color buffer, depth buffer, and stencil buffer. The drawing surface also details the bit depth of each of the required buffers. There are three types of drawing surfaces that can be created in EGL [23]. These are pbuffers, pixmaps and windows. Windows are attached to the native window system and are rendered to the graphics memory. Both the pbuffers (off-screen rendering to user memory) and pixmaps (off-screen rendering to OS native images) are pixel buffers that do not get displayed, but can be used as drawing surfaces [22].

EGL is needed to query the displays available on a device and initialize them. For example, within a car there might be several LCD panels and it is possible that we can render to surfaces, using OpenGL ES, that can be displayed on more than one panel.

The EGL API implements the above features and also further functionality, for instance, support for multiple rendering contexts in one process, power management, sharing objects (e.g. textures or vertex buffers) amongst rendering contexts in a process. The latest available version is EGL 1.4.

2.2.6 Conclusions

The usage of 3D graphics in embedded and small scale systems are growing substantially in all areas of life. One of the reasons to the increasing interest for graphics in embedded systems is the advances in display technologies. OpenGL ES and Mobile 3D Graphics are the two major platform independent graphics APIs. Mobile 3D Graphics actually works as a layer between Java and OpenGL ES. You make Mobile 3D Graphics calls from Java, and Mobile 3D Graphics invokes, in turn, OpenGL ES to manage the graphics rendering.

OpenGL ES is a low level API intended for applications written in C. OpenGL ES is as a matter of fact the standard for embedded applications. It is a main standard in the embedded avionics market due to its power and cross platform nature. It is a subset of desktop OpenGL, which con-

stituted as a benchmark when designing OpenGL ES. The full OpenGL library contains too many features that are not applicable for systems with restricted resources like embedded systems. An example is that calculations in OpenGL depends to a great extent on floating point arithmetic, but the support for floating point arithmetic is often limited or even missing on mobile devices.

Mobile 3D Graphics is an optional high-level 3D API for mobile Java that facilitates the manipulation of 3D graphics elements on mobile devices. When designing Mobile 3D Graphics the major goal was to overcome the performance penalty caused by Java virtual machines. The overcome of the performance penalty was fulfilled by raising the abstraction level of the API over that of OpenGL ES, allowing the applications to reduce the amount of Java code required for regular animation and rendering tasks. Mobile 3D Graphics uses the same rendering model as OpenGL ES.

There are a number of operating systems that allow end users to install applications written in C, but that number is small compared to those who support installation of Java applications. Java is supported by many different devices and operating systems.

Embedded software is very dependent on the hardware of a product. When rendering graphics the most demanding operation is floating point calculations. Though software implementations are possible, dedicated hardware acceleration provides better performance. Dedicated graphics hardware can also provide faster execution and lower power consumption.

EGL allows interfacing and control of resources such as frame buffers, memory, window and platform resources. There are two drawing surfaces that are rendered off-screen. Reducing the unnecessary use of those contribute to reduced use of system resources.

Should the choice be made between the two graphics libraries, the given choice is OpenGL ES. This is due to the fact that OpenGL ES is designed for applications written in low level languages (like C). Mobile 3D Graphics is designed, in part by the same developer, to supplement the OpenGL ES. If it is possible to use both graphic libraries it would give a wider range of applications, since Java is often used in areas such as mobile telephony.

2.3 Evaluate existing CAN protocol header

In the following sections, a brief description of what CAN be used for and a basic description of how the order is determined when several signals are sent simultaneously on the CAN bus is done. The information is the basis for understanding two problems in the existing design of the CAN protocol header.

In the existing solution the data length of the CAN signal has the biggest impact on what priority the signal will get. An additional problem is that

the signal number assigned to the signals does not follow any pattern.

Finally, a design proposal will be provided in which the above problems do not exist. A conclusion of the section is also given in the end.

2.3.1 Background

Many vehicles contain several embedded computers. The increase of electronics within the automotive sector is the result of customers wish for greater comfort and better safety and also a result of organizations requirements for improved emission control and reduced fuel consumption. The latter is also an important aspect for vehicle owners.

CAN (Controller Area Network) is a vehicle bus standard designed to allow devices to communicate with each other within the network without a host computer. On a CAN-bus, data is sent and received using message frames. The message frames carry data from the sending node to one or more receiving nodes.

When a CAN device transmits data onto the network, a unique signal identifier precedes the data. The signal identifier does not only hold information of the content of the message, but is also used to set the priority of the message. In real time processing, requirements vary on whether a message is sent fast or not. Some parameters change rapidly, while other barely changes at all and this explains why all messages do not have the same priority. Assigning priorities to the messages is a way to increase the chance that all messages are delivered in time. The lower the numerical value of the binary representation of the message frame is, the higher the priority[19].

In the occurrences where more than one node wants to transmit information at the same time, a non-destructive procedure guarantees that messages are sent in the correct order and that no information are lost. The highest priority ID message is sent first and all the lower prioritized nodes automatically becomes receivers of the highest priority message and retry to send after the highest priority message was sent. The priorities of the messages are assigned during the initial phase of system design and can not be dynamically changed.

This operation can be described, a little simplified, as if a node wants to send a message to one or several other nodes it waits for the bus to be available (Ready). The message is sent and the priority of the message is compared towards other contingent signals (Send message). The signal with the highest priority continues to send and the other nodes automatically become receivers of the message (Receive message). Every node that received the correct message performs a control to determine if the message was intended for that node (Control). If the message was of use for the node it gets accepted (Accept), otherwise it is ignored[9]. See figure 2.5

There are two permitted sizes of the data frame. To get the two formats

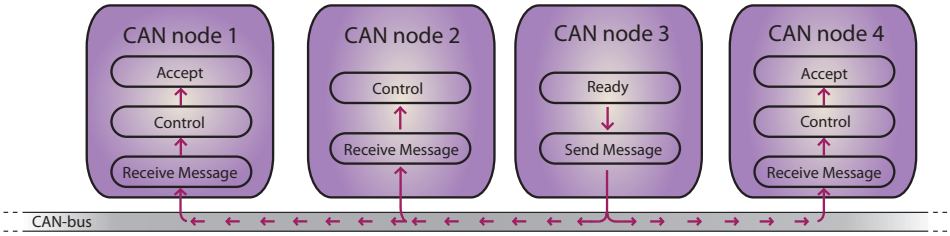

Figure 2.5: Broadcast send and receive by CAN nodes

to work together on a single bus it is decided that the shorter message type has higher priority in case of a bus access collision if signals have the same identifier.

In addition to the data frame, the CAN standard also specifies a remote frame. The remote frame holds the ID, but no data. A CAN device uses the remote frame to request information from another device to transmit the data frame for the ID. In other words, the remote frame supplies a way to poll for data[19].

2.3.2 How it looks today

In a CAN network, the messages transferred across the network are called frames. ENEAs CAN protocol supports two frame formats, the essential difference being in the length of the data frame. In the short frame format 8 bits of data is permitted and the long frame format need several signals in order to be completely sent. Figure 2.6 shows the fields of the frame formats, and the following sections describe each field. The CAN protocol header that is used in DySCAS today is 29 bits long, in accordance with the used Microcontrollers manual[15].

A main difference in ENEAs structure of the CAN protocol header is that a bit (priority) that resolves the length of the data frame precedes the signal ID, that otherwise determines the priority of the signal.

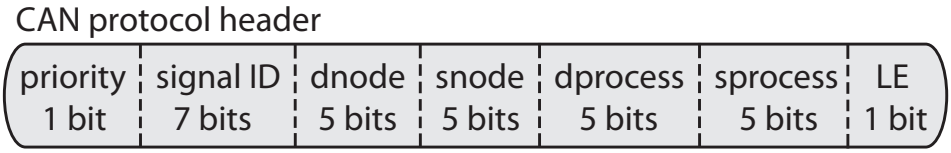

Figure 2.6: CAN protocol header as it looks today

priority The priority bit differentiates short frames from long frames. Because the priority bit is dominant (0) for short frames and recessive (1) for long frames, short frames have always higher priority than long frames.

signal ID The signal ID field contain the identifier for a CAN frame. Both formats (long and short) have a 7-bit field (7 bits = 128 different signals). In both formats, bits of the signal ID are transmitted from high to low order.

dnode Destination node (5 bits = 32 different nodes), according to IEEE a CAN bus usually consists of 30 nodes.

snode Source node (5 bits = 32 different nodes), which means 32 nodes can communicate on CAN bus.

dprocess Destination process

sprocess Source process

LE The endian bit decides the byte order. There are two choices in this protocol; big or little endian. Big endian (0) stores the most-significantbyte first (at lowest address) and little endian (1) stores the least-significant byte first. Big endian is most commonly used.

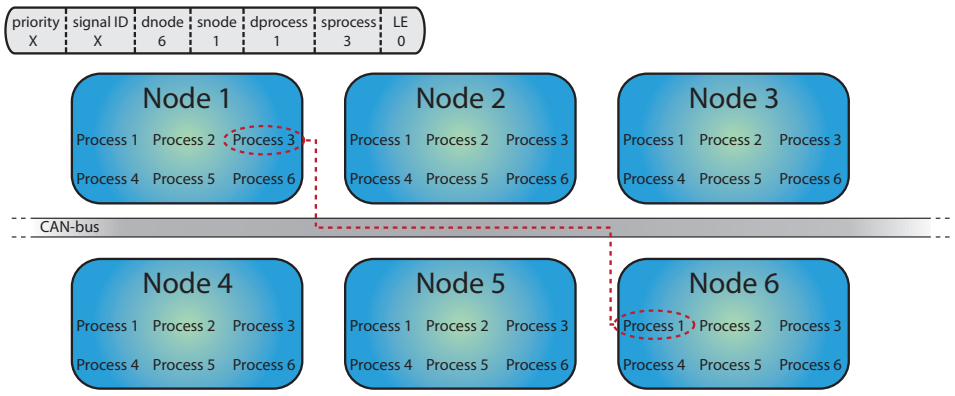

Figure 2.7: Send any signal from process 3 in node 1 to process 1 in node 6

Figure 2.7 exemplifies how the protocol is used. The fields marked with X can have any values that correspond to the description of the protocol.

As described earlier the CAN protocol must implement a bus allocation

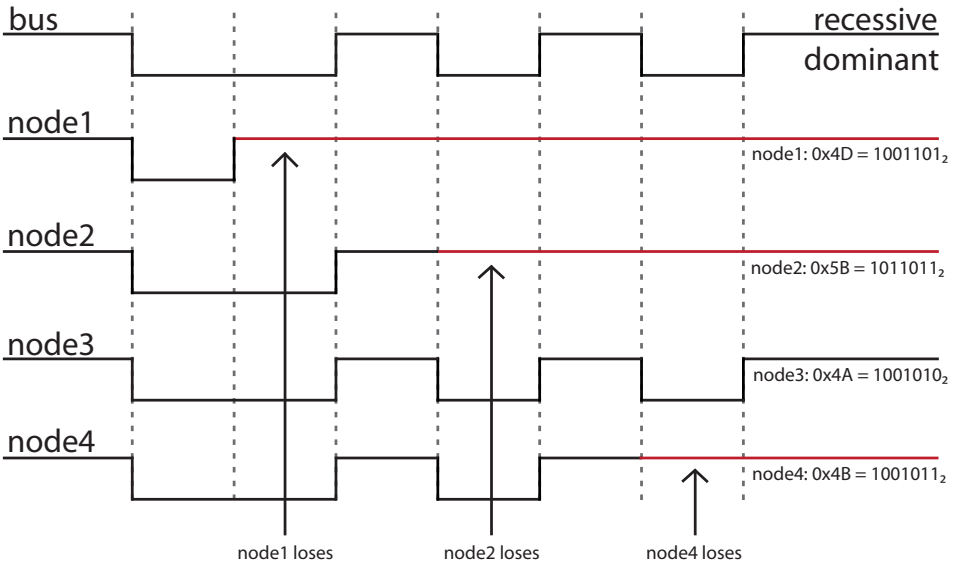

Figure 2.8: Example: How the non-destructive bitwise arbitration works

method that guarantees that there is always explicit bus allocation, even when there are instantaneous bus accesses from several nodes. This procedure decides what signal that has the highest priority and consequently gets to send its message. CAN use the established method CSMA/CD (Carrier Sense Multiple Access with Collision Detection). In the DySCAS project it is used with an enhanced capability of non-destructive bitwise arbitration to grant collision resolution, hence CSMA/CR (Carrier Sense Multiple Access with Collision Resolution).

The method of non-destructive bitwise arbitration uses the CAN protocol header of the message to resolve any collision by the means of a comparison of the binary representation. This is achieved by CAN transmitting data through a binary model of "dominant bits" and "recessive bits" where dominant is a logical 0 and recessive is a logical 1. When a comparison is made the 0 has higher priority than the 1, just like using a logical AND gate. This addition also contribute to a more efficient use of the CAN bus.

Figure 2.8 shows four CAN nodes attempting to transmit messages, using identifier 5B hex, 4D hex, 4A hex and identifier 4B hex respectively. All messages is presumed to have the same length (or value on priority bit). As each node transmits the 7 bits of its signal identifier, it checks the bus to find out if a higher priority identifier is being sent simultaneously. If an identifier collision is detected, the losing devices directly stop sending and wait for the higher priority message to complete, before automatically retrying. Since the highest priority signal identifier continues its communication without interruption, this scheme is referred to as non destructive bitwise

arbitration. This ability to resolve collisions and continue with high priority transmissions is one feature that makes CAN ideal for real-time applications.

2.3.3 Problems in the implementation

The problem with the implementation that is used today is that the 8 most significant bytes in the message frames are not set in a well thought out way. There are basically two main problems in the implementation:

1. The priority bit has the greatest impact of the message priority
2. The signal IDs are not assigned in a pre-defined manner

False outcome can be obtained if the priority bit will determine the order between two signals. This could, for example, lead to that a short signal that only update somewhat irrelevant is sent before a long, but important, signal from the system level. The signal IDs are not assigned in a pre-defined manner that would benefit important and commonly reappearing signals. The signal IDs has simply got the first available value and has consequently not the correct intergroup order. Signals from applications and the system level is appearing in a mixed scheme.

These shortcomings have effect on the order in which signals are sent according to the non-destructive bitwise arbitration.

2.3.4 Mapping of the signals

A first step to overcome the shortcomings, and to assign the signals more accurate priorities, was to identify the signals used in the system. This was done by looking through all signal files and examines a log file from when the system was up and running. Subsequently, some signals were removed, because they were not used.

Of the remaining signals a division has been made into two groups; system signals and application signals. All signals have change priority, but the application signals have not changed priorities in relation to each other, because they are rarely used simultaneously and thus will not "compete" for access to coach against each other.

The system signals have been analyzed and given priorities accordingly to its relative function in the system and how often it is sent. The signals used in the start up of the system are generally assigned a high priority.

The total number of signals retained in the system is 64 system signals and 27 applications signals which can be either long or short.

2.3.5 Design proposal

Since the weaknesses in the old CAN protocol was located in the first two fields of the header frame, these were investigated in order to assign more

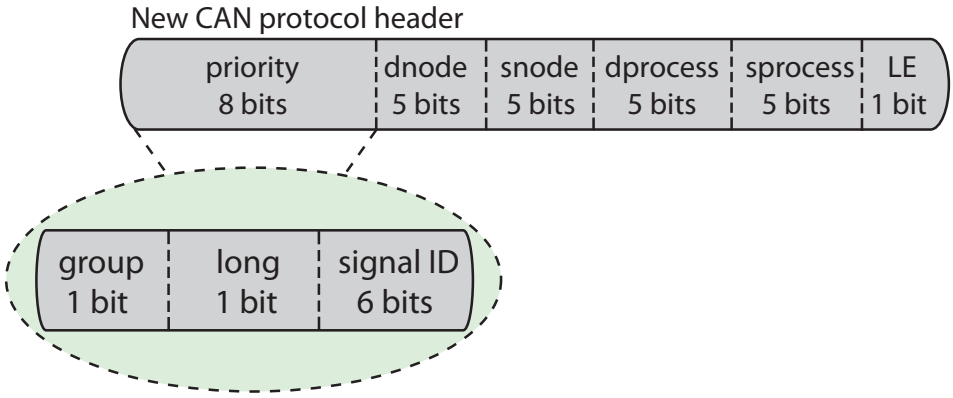

Figure 2.9: An enhanced CAN protocol header

accurate priorities to the signals. The main goal of the restructure was to reduce the significance of the length of the data frame and to assign higher priorities to system level signals relative application signals.

Figure 2.9 shows the proposed CAN protocol header. The old CAN protocol header used 8 bits (1 bit + 7 bits) for priority and signal ID. These 8 bits was divided into three parts in the new design; group, long and signal ID.

"Group" determines if the signal is of system or application type. The group bit of system signals are assigned to be dominant (0), while the group bit for application signals are assigned as recessive (1).

The "long" bit has the same functionality as the prior priority bit; hence it resolves the difference between short and long signals. To place this bit after the group bit has made that system signals more prioritized than the application signals. Information that takes short time to send shall also have a higher priority than signals that need long time to complete within the two groups. This is achieved in the new design.

The "signal ID" also has the same meaning as the earlier used signal ID, except that it only is of 6 bits length. This allows up to 64 different signals within each group that can be both short or long signals.

2.3.6 Conclusions

CAN is used in the automobile sector as a mean of communication between electronic control units and use priority to determine the order in which signals will be sent. The existing solution within DySCAS is based on that the signals length has the biggest impact on the priority. Short signals are considered to be more important than long signals and shall have a higher priority.

In the old solution, there is no structure in the choice of signal ID, since

the signals have gotten its signal ID in the order they were created. A signal that was created in a late phase of the development has got a lower priority because of this. These are the problems with the old solution of the CAN protocol header.

Based on the new structure of the CAN protocol header, signals will be given a more correct order in the priority determination. It means that the system signals will always be considered more important than the application signals. Short signals will still be given a higher priority than long signals.

2.4 Prerequisites for GPS antenna driver

2.4.1 What is GPS?

GPS (Global Positioning System) is currently the only available applicable system for satellite navigation. A set of more than 24 satellites are in use today that gives everyone with a GPS receiver the possibility to determine their position (longitude, latitude and altitude), regardless of weather, time of the day and location around the world. GPS is a necessary resource in navigation on land, ocean and in the air. The GPS system can determine your location anywhere on or above the earth with a little error margin. The system can obtain even better result by using information calculated by special GPS receivers at known fixed location around the world.

GPS has also become an important and useful time reference in our society, as the signal sent from the satellites hold time information from embedded atomic clocks.

Calculating the position

The basis for positioning determination is to calculate time differences for signals sent from satellites to receivers, which in turn are converted to distance. The satellites send information of the time and date, its satellite ID and details about where, in their orbit, it is to let the receiver compute their position.

If a receiver gets information from one satellite, it can compute the distance to that satellite and consequently know that it is somewhere on the surface of an imaginary sphere with equal radius to the calculated distance. If the receiver gets information from one more satellite it can determine its position to be somewhere on the line where the two spheres are intersecting each other. Information from a third satellite let the receiver determine its latitude and longitude, while it is required information from four satellites to be able to decide at what height you are (altitude).

World Geodetic System 1984 (WGS84) is the coordinate reference frame of geographical coordinates that is used by GPS. It is a reference frame for

the lines of latitude and longitude when producing maps.

Possible Use

The most obvious uses of GPS receivers, based on the description of how it works, are to exploit the accurate time information that the satellites send out and to use the information of where you are. The second use case provides, together with map software, driving directions to the driver. You can also get information about the surroundings by combining this information with a GIS (Geographical Information System).

The GPS receivers commercially available today have the drawback of to a great extent being static. In the DySCAS project there are possibilities to use the GPS in a larger extent. The greatest possibilities to broaden the use of GPS receivers are to combine them with other technologies.

Please note that the possible uses proposed are outside the scope of this thesis. This thesis will focus on providing working device drivers, pick out simple information from the GPS receiver and forward the information to the rest of the system.

Accidents Since mobile phones are possible devices to attach to DySCAS, this opens up great opportunities to transmit information via an interface already used by many people daily. Mobile phones can be used with GPS tracking devices in security purposes. Many vehicles already have GPS receivers and in case of an emergency the information can be used to inform the emergency personnel about where the accident happened. The GPS information can also be used by the medical staff to determine the ambulance nearest to an accident site, enabling the quickest possible response in life-threatening situations.

Planning the trip Often when you plan your travels you are not in the vehicle. The ability to connect additional devices to the vehicle in DySCAS allows you to use a portable GPS or a cell phone equipped with GPS. You can plan your trip at home and transfer the travel information when you get to the vehicle.

Travel information The time information is very useful in the public transport system. Many companies already use it and it gives information about when, for example, the bus will arrive to the next station. If there are any delays, that information can spread quickly.

2.4.2 Conditions in this project

Hardware prerequisites

The GPS receiver used in this project is integrated on the development board. The hardware is a micro controller from Freescale, MC9328MXS[11], based on an ARM9 core. The GPS receiver module on the development board is of brand u-blox, TIM-LA[27]. It supports both active and passive antennas and it provides two serial ports and seven GPSMODE pins to offer boot-time configuration capabilities.

Configurations of GPS receiver The values of the configuration pins at start up will determine the settings of the GPS immediately after startup. After boot-time these settings can be modified using the UBX configuration messages (see *Communication interface*). The settings set through UBX messages will remain until the GPS is reset.

Two of the GPSMODE configuration pins are used to determine what operating mode the GPS will use. Hence, a total of four possible choices can be made. The default choice is "Continuous Tracking Mode" (CTM), which is configured for optimal position accuracy. Even though the other configurations are more power saving, the default mode will be used, at least initially.

Two of the other configuration pins are used to set the update rate of the CTM. The default value is an update rate of 1 Hz, which will be kept.

Finally, three pins are applied to determine the baud rate and output message set. The default baud rate is 9.6 Kbaud and the supported NMEA sentences are GGA, RMC, GSA, GSV, GLL, VTG, ZDA, TXT. The baud rate will however be changed due to that another GPS is implemented for middleware SHAPE and adjustments will be made to send information in a conformably way. The advantage is that they then can use a similar COM handler (see *Connection from processor to system*), which receives information from the GPS and forwards this information to the application layer for analysis. The COM handler is located in a layer between the device drivers and middleware SHAPE. The other GPS use a baud rate of 19.2 Kbaud and the NMEA sentences GGA, RMC, GSA and GSV. At the time of writing the information of interest from the sentences is time, position and date. With the settings of this projects GPS giving a baud rate of 19.2 Kbaud, a few more sentences (compared to the default settings) are provided; GRP, GST and three proprietary sentences.

Except for the configuration pins there also other pins on the GPS module that are directly connected to the processor. One of them is the GPS_RESET. By setting the signal GPS_RESET high, the GPS module is put in sleep mode for lower power consumption. Another pin of the GPS module directly connected to the processor is the TIMEPULSE, which

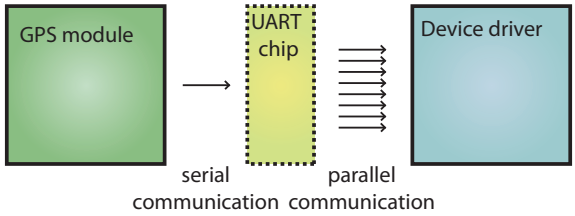

Figure 2.10: Communication from GPS to device driver

is an entirely output data pin. The GPS receiver provides a hardware-synchronized time pulse with a time pulse period of 1ms to 60s. Change of settings for the TIMEPULSE is made through UBX proprietary messages. The update rate (mentioned earlier) and the TIMEPULSE must be configured to result in a meaningful pattern. The default setting for the TIMEPULSE is 1 period per second and the pulse length is 100ms. This setting will be kept.

Connection from GPS to processor

The hardware is equipped with an external (to the processor) UART chip with four UARTs. The UART chip is a Philips SC16C554. The Philips SC16C554 is a 4-channel Universal Asynchronous Receiver and Transmitter (QUART) used for serial data communications. Its principal function is to convert parallel data into serial data and vice versa. The hardware's internal GPS chip is connected to UART4 of the quad UART and a full 9-pin RS232 interface is connected to the UART1 of the Quad UART. A working driver for the external UART should be available in the board support package (BSP) that the work is based on.

The Philips SC16C554 is connected to the processor (see Figure 2.10) with an eight bit parallel data bus and a five bit address bus. The chip contains an address decoder that controls the chip select signals of the UART controller so that the UART register can be accessed according to the registered memory allocation for the controller. The total memory for the quad UART controller is separated into four equally large parts. Each of the four UARTs has an interrupt signal connected to the processor. Configuring of the memories and interrupts are performed by the bootloader, but the operating system needs to be configured the same way. This is done in a configuration file before the system is booted. The bootloader is software stored in the flash memory that starts up when the system is booted. Sometimes the bootloader is used to boot the operating system and to setup the system.

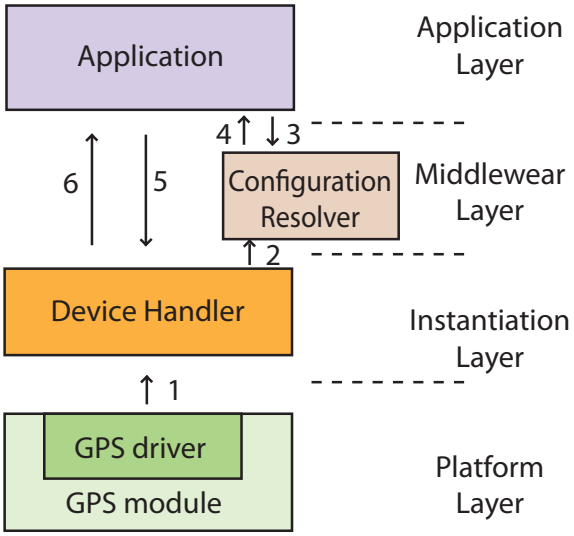

Figure 2.11: Sequence of the registration of a service and the initialisation of the communication between the application and the Device Handler.

Connection from processor to system

A Device Handler operates the communication with external devices to the DySCAS system. A separate Device Handler is needed for all physical ports or any other incoming or outgoing data. The structure of a Device Handler does not differ that much and small modifications is needed to be made to create a new Device Handler to another port.

The Device Handler registers its services to the platform independent middleware layer, which in turn offer these services to applications requesting for them. Since most services can only be used by one application at a time the middleware manages to decide what application to use the Device Handler. This is not a limitation in middleware, but a physical limitation in many ports. It would probably be possible for several applications to subscribe to a handler that is connected to a sensor, but it does not work that way in the present solution.

The application requesting a service via the middleware will be linked with the correct Device Handler, and can after that initiate a communication directly to the Device Handler. In SHAPE the communication take place in the form of sending signals. At the first contact from an application the Device Handler will save a reference and will further on send all incoming transmissions from its device to that application. At the first contact the application also sets the parameters that will be used in the succeeding communication.

The existing system already has a Device Handler for the COM port,

which uses functionality from the built-in UART control of the system hardware. An exact copy of this Device Handler can be used or adjustments can be made to fit the GPS that will now be added to the system (see Figure 2.11).

The Device Handler for the COM port is divided into two parts; an Interrupt Handler and the ordinary COM Handler. The Interrupt Handler receives an interrupt from the hardware when a new byte of information is available. The Interrupt Handler then collects the data and sends it to the COM Handler. The COM Handler receives the byte and sends the character to the application that has requested its services. If the COM Handler receives a byte, but has no subscriber of its services, it will throw the character away. For the COM Handler that is in use, a transfer rate of 19200 baud is used. The analysis of the data is made in the applications to collect the information that is wanted. One way to extract information from the NMEA messages is presented later in this chapter.

Communication interface

The GPS that will be used in this project comes with a highly flexible communication interface. It supports the NMEA (National Marine Electronics Association) and the UBX[10] binary protocol and is able to accept differential correction data (RTCM). The GPS is multiport capable, which means that each protocol can be assigned to several ports at the same time with different settings per port (baud rate etc.). It is also multiprotocol capable, which means that more than one protocol is possible to assign to a single port. This is especially helpful for debugging purposes.

The UBX binary protocol can set and poll all the available actions and messages from the GPS. Using asynchronous RS232 ports, the GPS communicates with a host platform in terms of the alternative, UBX protocol, to carry GPS data. This proprietary protocol has the following key features:

- Compact. 8 bits binary data is used
- Checksum protected, using low-overhead checksum algorithm

The NMEA standard uses a simple ASCII[11], serial communications protocol that defines how data is transmitted in a "sentence" from one transmitter to one receiver at a time. The UBX protocol offers a greater flexibility and more powerful messages than NMEA protocol, since it is based on an efficient binary data handling. At the same time the UBX protocol is more difficult to use for the same reason.

UBX protocol will mainly be used to configure the GPS receiver, while the NMEA protocol will be used for the outgoing communication in the

[10] UBX: u-blox proprietary protocol
[11] American Standard Code for Information Interchange

demonstration platform. This is partly because the NMEA protocol does not take advantage of all the features of the GPS due to bandwidth limitations of the ASCII-protocol. A short presentation of the fundamentals of the NMEA protocol follows in the next section.

NMEA protocol

The NMEA protocol is an industry-standard protocol that was developed for marine electronics and was originally designed to allow data exchange between various sensors and navigation equipment aboard ships. Nowadays, it is a de-facto standard for GPS receiver data output[27]. The protocol expresses the data in the format of ASCII. This is a standard format for GPS applications.

The GPS supports NMEA-0183 version 2.3, which is upward compatible with version 3.0. NMEA consists of sentences, of which the first word is called a data type. The data type defines the interpretation of the rest of the sentence. Each data type have its own unique interpretation and is defined in the NMEA standard. Sentences start with a $ and end with carriage return/line feed. GPS specific messages all start with $GPxxx where xxx is a three-letter identifier of the message data that follows. NMEA messages have a checksum, which allows detection of corrupted data transfers. Below is a list of some of the NMEA standard sentences the GPS support:

- DTM - Datum Reference
- GBS - GNSS[12] Satellite Fault Detection
- GGA - Global Positioning System Fix Data
- GLL - Geographic position - latitude/longitude
- GSA - GNSS DOP and active satellites
- GSV - GNSS satellites in view
- RMC - Recommended minimum specific GNSS data
- VTG - Course over ground and ground speed
- GRS - GNSS range residuals
- GST - GNSS pseudo range error statistics
- TXT - Text messages
- ZDA - Time and Date

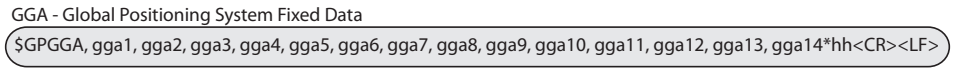
GGA - Global Positioning System Fixed Data
$GPGGA, gga1, gga2, gga3, gga4, gga5, gga6, gga7, gga8, gga9, gga10, gga11, gga12, gga13, gga14*hh<CR><LF>

Figure 2.12: Position fix related data such as position, time, number of satellites in use, etc.

Parameters	Descriptions	Notes
gga1	UTC time as position is fixed	hhmmss.ss, hh: hour; mm: minute; ss.ss: second
gga2	Latitude	dddmm.mmmmm, dd: degree; mm.mmmmm: minute (0 ° - 90 °)
gga3	Latitude sector	N: North; S: South
gga4	Longitude	dddmm.mmmmm, dd: degree; mm.mmmmm: minute (0 ° - 180 °)
gga5	Longitude sector	E: East; W: West
gga6	GPS quality indicator	0: No fixed or invalid position 1: SPS Position available 2: Differential GPS (SPS) 6: Estimated position (DR)
gga7	Number of satellites used in position estimation	xx: 0 - 12
gga8	Horizontal Dilution of Position (HDOP)	xx.x: (00.0 - 99.9)
gga9	Altitude above mean sea level (geoid)	
gga10	Unit for Altitude	M: meter
gga11	Geoidal separation	Difference between WGS-84 earth ellipsoid and geoid
gga12	Unit for geoidal separation	M: meter
gga13	Age of differential corrections	unit : second; null when DGPS is not used
gga14	Reference station ID (DGPS)	xxxx: (0000 - 1023)
hh	Checksum	hex number (2 - character)
<CR><LF>	End of message	

The GGA sentence in Figure 2.12 shows an example that provides essential fix data. This sentence is of biggest interest for this thesis, since it

[12]Global Navigation Satellite System

gives most information needed to accomplish the task. The most important NMEA sentences except for GGA is the RMC, which provides the minimum GPS sentences information, and the GSA which provides the Satellite status data.

Pick out data from protocol

The NMEA 0183 standard specifies how the output is formatted for GPS data output, usually on a serial port. The standard is a collection of variable length strings called sentences delimited by end of line markers. The standard sentences have well-defined syntax and semantics. The NMEA specification is somewhat open and allows small variations in the implementation, but it always demands commas as a separator between NMEA data fields. A problem to access the correct data from the protocol is that the sentences are sent as a stream. This will be discussed later on.

Each sentence is roughly made up of three parts; identification, data, checksum. The data is, as mentioned above, separated into data fields by commas and is holding the information that is of most interest. If data for a field is not available, the field is simply omitted, but the commas that would delimit it are still sent, with no space between them. A detailed description of how sentences generally are structured will be presented below.

What does the stream look like? As mentioned, the sentences are streaming from the GPS receiver via the serial port. Each sentence starts with a "start of sentence"-character, "$", a two-character prefix that identifies the type of the transmitting unit, a three-character identification of the sentence, followed by several data fields separated by commas, and ends with a optional checksum and a two-character end of sentence (see below).

The format for standard sentences may be defined by the following production rules[24]:
Sentence: "$" <prefix> <sentence ID>,<data fields>*<checksum> <eos>

1. $: Start of sentence
2. prefix: two characters that identifies the type of the transmitting unit. The most commonly used is "GP", identifying a generic GPS. These[13] are the only sentences that can be transmitted by a GPS receiver.
3. sentence ID: three characters giving information about what type of sentence it is. Examples have been given in 2.4.2.
4. data fields: the data fields are the biggest difference between different sentences. They hold information interesting to the users of GPS receivers (e.g. position, speed).

[13] plus some proprietary sentences depending on the GPS receiver

5. checksum: is a hexadecimal string formed as a bitwise XOR of all characters between the "start of sentence"-character and the asterisk (see above in production rules).

6. end of sentence: two-character end of sentence/string, <CR> <LF>

Writing a parser To be able to read the NMEA sentences you will have to have some sort of functionality that is able to receive the serial data stream and analyze it. First it has to decode the stream into sentences, by finding the "start of sentence"-character. Secondly it has to parse the sentences for their content and finally convert the content into something understandable by the user, which is not a trivial task.

The second step indicates that there is a need for a parser that searches and extracts the data fields. This can be done by keeping track of the character position within the sentence. An even better solution is to extract everything between two commas. Since some data fields are of variable length or are even empty, this solution makes the parser independent of the number of digits for the individual data field. It will even be able to handle empty fields ",,". Beside the commas, there are several ways of giving the parser more functionality. Additional functionality can be introduced to error check the sentences (e.g. by validating the checksum). If any error is found, the NMEA parser should reject data output by the GPS receiver and by no means forward it to, in this theses case, the application.

Design choices When designing a NMEA parser you are encountering several choices that will affect the functionality of the parser. The NMEA standard supports a wide range of standard sentences, and there are also opportunities to define own structures of sentences (proprietary sentences). As mentioned in 2.4.2 a GPS receiver can only send a limited number of these sentences, but it is still a large amount of different sentences. When you write a parser you must choose between making a large complex parser or to limit the compatibleness of the parser. Moreover, perhaps you need to take into account if the parser will be used for more than one GPS recipient, which may lead to that there is not the same support in all recipients.

Since the objective for this master thesis lies within programming device drivers and to write a simple application that connects my task areas, a parser with limited support for different sentences will be designed. There will be no error checks on the messages and hence no error correction.

State-machine Variakojis[28] suggests a NMEA parser implemented in C++. In the paper he uses a state machine 2.13a that tokenizes the first character of a sentence, the checksum, and the end of sentence. Based on his state machine I have designed a simplified state machine 2.13b, which complements the parser proposal which is provided in 2.4.2.

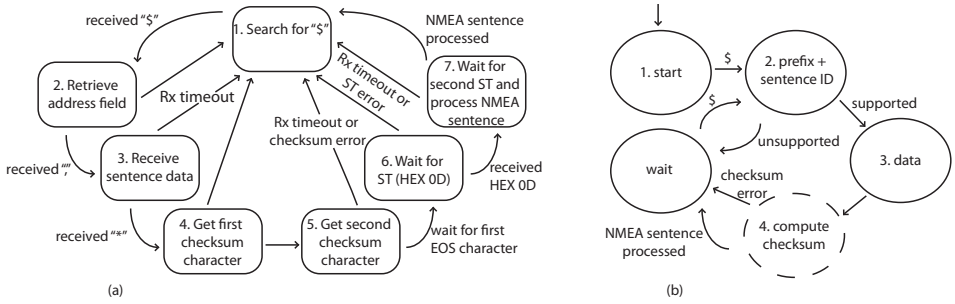

Figure 2.13: (a) Variakojis state-machine (b) My simplified state-machine

To track the structure of the NMEA data you can use a state machine. The state machine might be formed to look for dollar signs, records characters until it sees a comma, decides if the sentence is interesting, if it is it records characters until the end of the line, then validates the checksum and parses the fields it wants. If the sentence isn't one of interest, or if any errors occur, it goes back to waiting for a dollar sign.

Design proposal for parser The parser can be composed of two main parts. One function, messageSplit, which manages the breakdown of the sentence through the use of the comma character "," that exist between all the data fields. It presents the words as strings and keep track of whether there are more data fields or not.

The second function keeps track of whether there is support for the requested sentence. It receives the data stream directly from GPS receiver (as a call parameter) via the serial port and it will also be called with a "struct" parameter that can be used to store information. It uses the first function to read one data field at a time. By scanning the first field (e.g. $GPGGA) the function knows what sentence type the stream transmit and then (if supported) to overwrite/update all data fields. These will finally be converted to be readable and easily understood by humans. By using this method you are forced to make comparisons between your first field and strings in, for example, case or if statements.

Most of the work done by the parser consists of determining the sentence type and separating the different items. The parsed data are stored as strings in a struct. The parser can be extended to recognize if the data coming from the GPS receiver is valid by using the optional checksum.

2.4.3 Conclusion

The GPS system gives everyone with a GPS receiver the possibility to determine their position. The basis for positioning determination is, to calculate

time differences for signals sent from satellites to receivers, which in turn are converted to distances.

The most obvious uses of GPS are to exploit the accurate time information that the satellites send out and to use the information of where you are. The second use case provides, together with map software, driving directions to the driver.

In the DySCAS project there are possibilities to use the GPS in a larger extent. The greatest possibilities to broaden the use of GPS receivers are to combine them with other technologies.

A possible feature is to plan the trip at home by using another GPS receiver and to transfer that information when you arrive to the vehicle. Another use case is to predict how long it will take before the vehicle reaches a certain point.

The scope of use is more or less endless depending on the number of devices you can combine the GPS with.

The NMEA protocol will be used for the outgoing communication in the demonstration platform. It is an industry-standard ASCII-based protocol that defines how data is transmitted in a "sentence" from one transmitter to one receiver at a time. The most important NMEA sentences for this thesis is the GGA, which gives the position as coordinates in a longitudinal and latitudinal system, the RMC, which provides the minimum GPS sentences information, and the GSA which provides the Satellite status data.

2.5 Summary and Conclusions

Q1. What graphics library is appropriate to use in SHAPE?
OpenGL ES and Mobile 3D Graphics are the two major platform independent graphics APIs. Mobile 3D Graphics actually works as a layer between Java and OpenGL ES. OpenGL ES is a low level API intended for applications written in C. Mobile 3D Graphics is an optional high-level 3D API for mobile Java that facilitates the manipulation of 3D graphics elements on mobile devices. Mobile 3D Graphics uses the same rendering model as OpenGL ES.

Q2. What choices can be done to reduce needed system resources for graphics?
Embedded software is very dependent on the hardware of a product. By using dedicated hardware acceleration for floating point calculations and dedicated graphics hardware you can get faster execution and lower power consumption.

EGL allows interfacing and control of resources such as frame buffers, memory, window and platform resources. Reducing the unnecessary use of off-screen drawing surfaces contribute to better use of system resources.

Q3. What limitations and shortcomings imply communication through CAN in DySCAS?
The existing solution within DySCAS is based on that the signals length have the biggest impact on the priority. Short signals are considered to be more important than long signals and have higher priorities. The old solution has an insufficient structure of the signal ID, since the signals basically have gotten its signal ID in the order they were created.

The proposed structure of the CAN protocol header correct this by grouping the signals in two categories; system- and application signals. A review of the signal IDs have been made and they have gotten new values. Short signals are still considered more prioritized than long signals.

Q4. What features can a GPS contribute to DySCAS?
The most obvious uses of GPS are to exploit the accurate time information that the satellites send out and to use the information of where you are. The second use case provides, together with map software, driving directions to the driver.

The greatest possibilities to broaden the use of GPS receivers are to combine them with other technologies. A possible feature is to plan the trip at home by using another GPS receiver and to transfer that information when you arrive to the vehicle. Information from the GPS receiver can be used to predict how long it will take before the vehicle reaches a certain point. Another use case for GPS receivers are to facilitate rescue work in case of an accident, by informing the emergency personnel as to where the accident occurred. The rescue personnel can also use GPS receivers to determine which vehicles that have the shortest route to the scene.

The scope of use is more or less endless depending on the number of devices you can combine the GPS with.

Q5. How do you pick out data from the GPS stream?
The NMEA specification is not as strict as there are any requirements on the number of digits in a data field, etc. But there is always a decimal point between two data fields, which makes the easiest way to extract information from a NMEA message, is to write a parser to search for two decimal points and read out all data in between. This way the parser is independent of the number of digits for the individual message fields and can even handle ",,".

Chapter 3

Design & Implementation

3.1 Implementing graphics port driver

3.1.1 Used hardware

The work with the graphic drivers has been conducted on an i.MX31ADS Application Development System (ADS) from Freescale Semiconductor. The work is carried out with a BSP for Linux as a base. The ADS consists of a Base board, a CPU board, and an MC13783 board (see Figure 3.1). Peripherals, such as displays and sensors, are attached to the ADS via the Image Processing Unit (IPU) hardware. The IPU is built to offer video and graphics processing functions and to be an interface between the i.MX31 processor and video/still image sensors and displays. The IPU contains functionality that is redundant for the area which this thesis work is focused on. Such functionality is, for example, video and image data pre- and post processing like resizing and rotating images/videos and color conversion.

The ADS support several different types of LCD interfaces and the display that comes with development system is a dumb (memory-less) active LCD panel, Sharp LQ035Q7DB02. This is the display that has been used during the master thesis. The Sharp display has a synchronous LCD interface provided with scan control by the i.MX31. This is comparable to LCD interfaces of previous i.MX processors[12]. That the display is memory-less means that it has no own memory buffer to read data from, but must fully rely on a created framebuffer (See section 3.1.3) that is on the hardware RAM. It is a 3,5" Quarter VGA display, which implies a resolution of 240*320 pixels. It contains no integrated control circuit, but it has support for backlight control and has touch screen capabilities. The display is connected through a flat panel cable to the base board. The base board can be used with only the CPU board connected, but the touch screen controller is built into the MC13783 chip and therefore the MC13783 board is required for this function to operate.

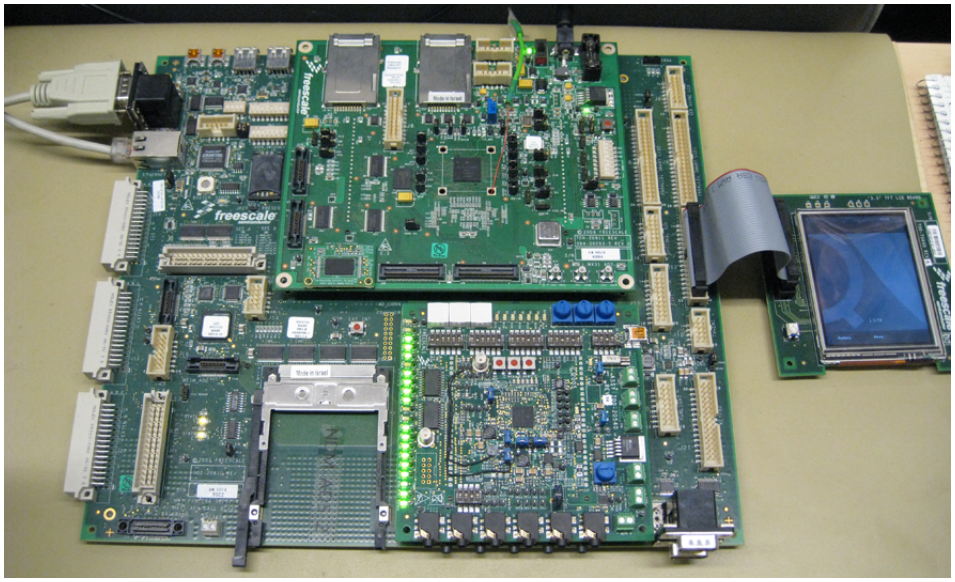

Figure 3.1: The i.MX31ADS hardware and Sharp display

3.1.2 Connection of LCD

As mentioned in the previous section, the display is of type memory-less LCD. This has some consequences. It has no on-board frame buffer and it is required to periodically be refreshed by the hardware. In this case that hardware is a LCD Controller (LCDC). The consequence of that the LCDC has to continuously send refresh signals to the display is an increase of the load on the CPU memory bus, if compared to a "smart LCD" that do not require periodic refreshing. When the refresh signals from the LCDC stops, the picture fades away.

The display is interfaced via VSYNC and HSYNC signals, where a signal is a bit. The VSYNC indicates the start of the next frame of data. A frame is the complete scan of the display. The HSYNC indicates the start of the next line of data. Figure3.2 shows how the display is connected to the baseboard.

The LCD Shift Clock (LSCLK) clocks the pixel data into the display drivers internal shift register. LSCLK is the output from the LCD Controller and must be less than 1/3rd of the high spped clock (HCLK) for TFT panels to ensure that data is output correctly from the LCD Controller. The HCLK is generated from the system phase-locked loop and the maximum value, 133 MHz, is provided by a division with a clock registry value, which is set by the user.

OE stands for Output Enable signal and is used to enable data to be shifted onto the display.

CONTRAST is used to control the LCD bias voltage to modify the

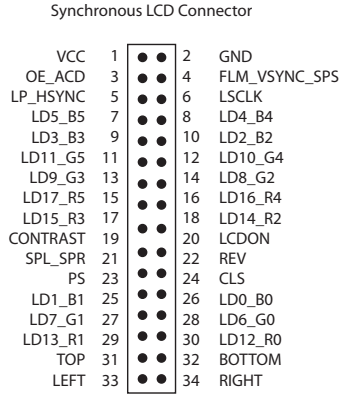

Figure 3.2: LCD Connector for e.g. Sharp TFT panel

contrast of the display.

The signals SPL_SPR, PS, CLS and REV is used for Sharp 320x240 high resolutions TFT panels only. Other displays that are attached to the port will not support these signals. SPL_SPR sets the horizontal scan direction. PS is a control signal output for source driver. CLS is the start signal output for gate driver. This signal is an inverted version of PS. Finally the REV signal is for common electrode driving signal preparation.

3.1.3 LCD Controller- and Framebuffer driver

Framebuffer device (often mentioned as framebuffer) is a concept that is related to the LCD controller for a display. The LCD controller driver and the LCD framebuffer driver interacts with the generic framebuffer driver, via function calls, in order to have hardware accessibility and to create an interface for software applications so it does not need to know about the lower levels. This two drivers may be, and often also is, made as only one driver.

The framebuffer can be seen as a memory buffer available for the LCD controller that contains data. Through the LCD framebuffer driver software applications gain access to graphics hardware and can indirectly talk to the display. Information sent to the framebuffer is in most cases data in the form of color values for each pixel. This information will then be redirected and displayed as an image on the LCD.

The framebuffer contains several structures that hold information about fixed, variable or alterable parameters that help the LCD Controller to read the data in the buffer in the correct way. Such parameters are for example "start of framebuffer", "resolution", "number of bits per pixel" and "details of each color in a pixel" (how to interpret the data and arrangement from most significant- to least significant bit e.g.)

Initialization of the drivers

The initialization of the system framebuffer starts when the kernel calls functions that configure the development board. During this phase all important functions start, including the framebuffer, which is the base for a working system. The framebuffer is booted by registering a device with information about the name, direct memory access address, size, etc. that can be read by drivers.

The next stage is to initialize the framebuffer driver. The driver will be registered by the system, but not tied to any platform at this stage. The necessary resources are started and structures applicable to the specific driver will be initialized.

Thereafter the setup of the IPU module is taking place. Information about the IPU's general features are registered to the system and a first configuration of the processes connected to it, like the synchronous display controller, is made.

When the IPU has been registered as a unit in the system and initialized correctly and undergone tests of the functionality, then the next layer of software that is related to the implementation of framebuffer (in this case the LCD framebuffer) to the i.MX31 is initiated and tested. The following initialization of the software is mainly based on function calls to functions connected to the IPU. The initialization process includes that the framebuffer gets registered to i.MX31ADS as a platform driver. A large part of the initialization process consists of tests like enabling channels, disabling channels, and execution of tests for interrupt request.

The driver for the i.MX31 was built with support for multiple displays, but if you want to hook up a display that has a different interface, you need to write a specific driver. The display drivers are very similar, so by looking at the existing driver you will get much of the work done. See Figure 3.3a for a graphical presentation of the initialization steps.

Figure 3.3b shows roughly the steps made by the LCD Framebuffer driver and the LCD Controller driver. The following steps are carried out by function calls within or to other functions. The LCD initialization registers the LCD driver to the platform. By making it a platform driver, pointers to required functions like probe, suspend or resume intended to this panel is set.

A central data structure needed in order to initiate a LCD display is fb_videomode. It contains information about the timing and configuration data for the panel, among other things, screen name, resolution, refresh rate and update information in form of time period between end of Vertical/Horizontal (V/H) Sync pulse and the start of the next active V/H time. Information from this and other structures are passed on to hardware registers and functions.

When the driver is installed, tests of the functionality (probe) is made to

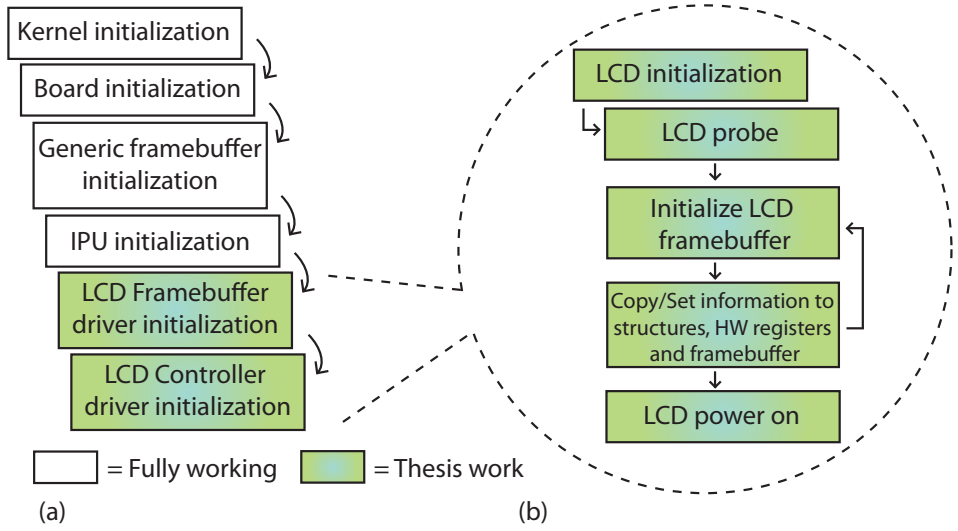

Figure 3.3: (a) Initialization on system level (b) Initialization on drivers level

take care of setting voltage levels, calls functions to initialize the LCD, turns it on and finally calls a function to notify the kernel that the installation of the LCD driver is done. The probe feature is a large and important part of the driver for the graphics port. In this function lots of function calls, to other parts of the system, has been gathered to together link the LCD framebuffer driver and LCD Controller driver together.

The next step is to initialize the framebuffer, which is made from the probe function. This step may need to be carried out one or two times depending on whether you want a single background graphic plane or if you also want a graphic window (also known as foreground plane). The LCD Controller includes two parallel data paths, one for the background plane and one for the graphic window. Data for the background and foreground planes are collected from system memory (e.g. RAM) via the LCD framebuffer driver into separate buffers. The LCD Controller will perform various actions, depending on how hardware registers are set. For example byte swapping on the data if a different endianness has been set in a hardware register. The LCD Controller has built-in hardware support for combining the two graphic planes. This is also used when an image is combined with a hardware cursor.

During the initialization of the framebuffer, information is copied and/or transferred between different structures (e.g. specific panel information) and information is set in the hardware registers. Important information is also written to the framebuffer. After this step the LCD is actually initialized. The last step is to turn on the panel and to register a client notifier. The

```
#define SIGNAL_NAME (number (e.g. 14))
struct signal_name
{
   SIGSELECT sig_no;
         U8      source_proc;
         U8      source_node;
         U8      dest_proc;
         U8      dest_node;
         ...
};
```

Figure 3.4: Example of a signal structure

notifier holds information such as a pointer to the function called when the installation of the LCD driver is done.

3.2 Adjustments of CAN protocol header

The adjustment of the CAN protocol header has been made for an existing CAN communication between several Phytec MM167 boards with Infineon C167processors on ENEAs own operative system OSE Epsilon. There is a total of six units connected to the CAN bus at the same time. The units are called nodes in the system. The platform where the communication takes place in is called SHAPE and this is ENEAs part of the project DySCAS. The development has been done in TASKING VX-toolset for C166/ST10, which is an embedded software development tool from the company Altium. In SHAPE the communication take place in the form of sending signals. The signals are composed of a signal number, an address header and data fields. As seen in section 2.3.2, the signal number and the address header is used in the CAN protocol header, while the data exists as a separate part of the protocol.

All fields in the signal are declared as being at least one byte, i.e. 8 bits. Figure 3.4 shows an example of a signal structure that contains some basic variables. The signal structures are in part structured in a consistent manner, but the data fields differ between the various signals.

One problem, that is the cause for that the optimization of the CAN protocol header is implemented, is that the space in the CAN protocol header is limited to 29 bits. These 29 bits you want to take advantage of in an appropriate way according to the scope of use of the system. After all, as new information appears, or if parts of the system is changed there might be a better use of these 29 bits available. Section 2.3.5 shows my design suggestion on how CAN header should look like. One thing, however, that

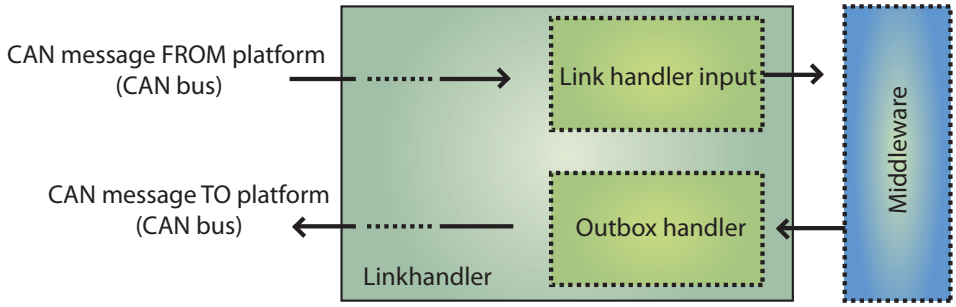

Figure 3.5: CAN messages sent to/from Linkhandler

should be stressed is that all units in the system uses the same endian and therefore the system does not use that bit. No tests are done to determine what type of endian the devices have.

However, the endian bit had to remain in the CAN protocol header for future use of the system, when devices with different endian could be used. This has no impact on the way of reading the CAN protocol header, since this part of the CAN message always is read the same way. The endian bit determines which way the following data should be read.

At start up of OSE Epsilon a number of prioritized processes are declared. The first process to be declared is the link handler, which gives that process the highest priority. The link handler manages the network communication and makes signals go from the source process to the destination process within or between two or more nodes. Every signal sent that is unknown to the system will be trapped by the link handler (a separate instance of SHAPE is available at each node) of that node. The link handler makes a decision based on the signal content, whether to forward the signal within the node or to send it to another nodes link handler. This fact makes it necessary that every signal has a unique signal number.

There are two registers that store the information for the CAN protocol header: upper arbitration register (UAR) and lower arbitration register (LAR). These registers are 16 bits in size each, of which 3 bits are reserved in the LAR. These 29 bits is what sets the limit on the CAN protocol header size. Figure 3.5 gives a simplified view of the parts affected by the CAN header protcol change.

3.2.1 Converting an OSE signal to a CAN message

As mentioned, all members of a signal structure is at least of size one byte (data fields might be bigger) and it is not until the signal is sent on the CAN bus the size is a problem. The link handler receive the signals unknown to the system and these signals arrive to the outbox handler. The outbox

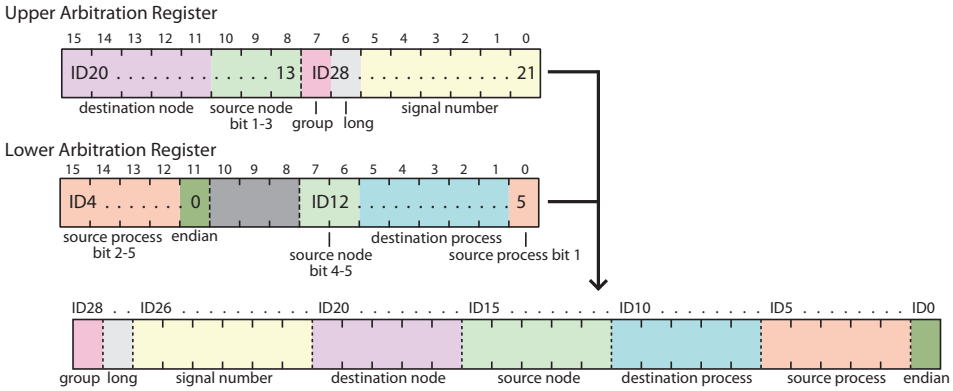
Upper Arbitration Register
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
ID20 13
ID28 21
destination node
source node bit 1-3
group
long
signal number
Lower Arbitration Register
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
ID4 0
ID12 5
source process bit 2-5
endian
source node bit 4-5
destination process
source process bit 1
ID28 . . ID26 . . . . . ID20 . . . . . ID15 . . . . . ID10 . . . . . ID5 . . . . . ID0
group
long
signal number
destination node
source node
destination process
source process
endian

Figure 3.6: Upper- and Lower Arbitration Register.

handler determine if the OSE signal is supposed to be sent forward to the destination. If so, the outbox handler has to convert the information in the OSE signal to fit a CAN message.

This is done by reading each part of the message (signal number, destination node, destination process etc. not the data) and to copy that information to the UAR and LAR.

Based on the number of bits that has been allowed for each field in the CAN protocol header (see section 2.3.5) a bitwise AND operation has been used to reduce the number of bits. The fields "destination node" and "destination process", which both are 5 bits long and stored with all bits in the same register, has been bitwise compared to 0x1F (=11111). The result from such a comparison is then shifted to the right position in the registers. The result from the example with "destination process" is left shifted 11 bits and stored in the UAR (See Figure 3.6). This procedure has been implemented for all fields in the order they appear in UAR and LAR.

To set the correct value on the "long" bit, the writing to the registers for all fields is made within an if-then-else statement. In order to separate the system and application signals, and thus provide the correct value to the group bit, the system signals got signal numbers within the range 0-63 and the application signals got values from 64 up to 127. To get the correct CAN signal number, the signal number from the signal structure was bitwise AND compared (as described above) with 0x3F (=111111).

When assigning a value to the group bit the following approach was used: Only the application signals would give 1 as a result when bitwise AND compared to 0x40 (=1000000), since that bit in decimal is the same as 64. The group bit was also left shifted one bit. This way the group bit was set, which resulted in that application signals also had a CAN signal number within the range 0 to 63.

3.2.2 Converting an OSE signal to a CAN message

The signals arrive to the link handler from interrupt processes. The purpose of the link handler input process is to assemble the CAN messages into OSE signals and to forward them to the middleware (a layer closer to the applications in SHAPE). A CAN message has the header information that the outbox handler has stored in the UAR and LAR. To transform a CAN message to an OSE signal is the same as doing the opposite of the steps in section 3.2.1. Figure 3.6 shows the UAR and LAR, where the information should be copy from. Figure 3.6 also explains how these registers are interconnected and the order in which the various bits in the registers are read on the CAN bus.

Since both the system- and application signals have signal numbers between 0-63 in the CAN messages, the group bit together with the signal number is used to recreate the original signal number from the OSE signal. A bitwise inclusive OR operation is made between the two bitwise AND operations. The result from the bitwise AND operation (group & 0x3F) is also right shifted 1 bit to give the correct value.

e.g. sig_no = ((group & 0x0080) >> 1 | (signal number & 0x3F))

3.3 Extend OSE 5 BSP with GPS support on i.MXS board

The work with the GPS antenna driver has been done on a custom-built hardware. The backbone of the hardware is a microcontroller from Freescale, MC9328MXS, and a GPS receiver from u-blox, TIM-LA.

The starting point for the work was an already written BSP for ENEAs operating system OSE 5.3. The BPS should be complemented by a driver for a GPS antenna. In addition to the hardware the GPS receiver is located on, a JTAG, ARM Multi-ICE, along with the software Real View Multi-ICE v.2.2.6 and Real View Developer Suite v.1.8.1, has been used. By using a JTAG you are able to connect to the hardware and execute, transfer or process code with the possibility to step through the executing code in steps and stop the whole system, which facilitate troubleshooting a lot. The Real View software's offers a compiler, trace software and supports debug possibilities.

The software TeraTerm was used to, via the serial port, load the bootloader onto the terminal. The bootloader in combination with the JTAG was in turn used to load the operating system to the hardware RAM memory.

The hardware is equipped with an UART chip with four UARTs. The GPS chip is connected to UART4 of the quad UART and the physical RS232

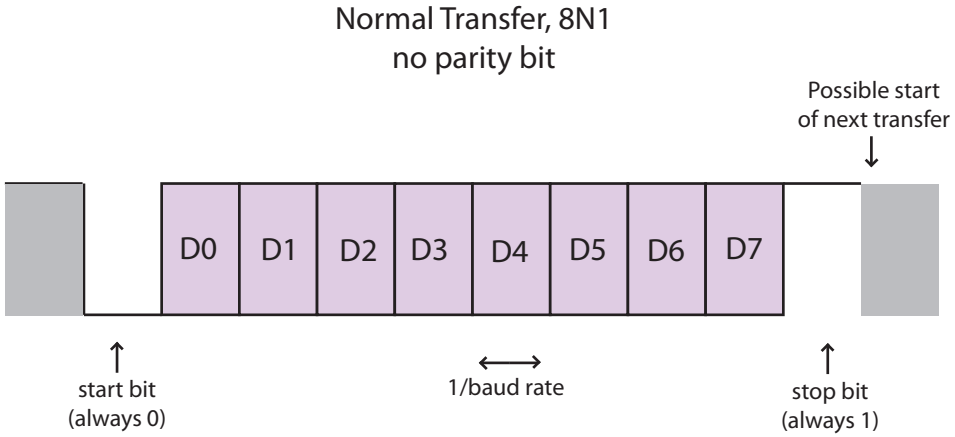

Figure 3.7: A normal data transmission of a single byte (8 bits data and 1 stop bit).

is connected to UART1 of the quad UART. The UART chip is connected to the i.MXS processor with an eight bit parallel data bus and each of the four UARTs has an interrupt signal connected to the processor.

The device driver will, from the GPS receiver via the external UART controller, receive data that are sent as one byte at a time (see Figure 3.7). This is a property defined by the NMEA and UBX protocols, not by the interface. The GPS receiver is connected to the i.MXS processor via UART4 of the external UART chip. Each of the four UARTs on the chip has an interrupt signal connected to the processor. Each time new data arrives aimed for the processor an interrupt will be sent and an interrupt process within the driver will allocate memory and collect the data. One of the features the UART chip has is to convert serial data to parallel data, which makes it easier to store the data in a signal (used in OSE) which is needed before the signal is sent to another process in the driver. This signal can either be analyzed or sent directly to UART1 of the external UART controller. This UART is connected to the physical RS232 port of the hardware. The UART chip will yet again convert the signal, but this time from parallel data to serial data. From this point the Device Handler takes on (see Figure 3.8).

There are two alternative ways of handling the communication between the GPS receiver and the device driver. Since it is a common UART (and it communicates with ASCII characters) connecting the GPS receiver to the device driver the usual way to handle this in OSE is to connect a standard driver (serdd.c) to the UART port, which is done for other UARTs to get an OSE standard interface to the serial port device. This will give you a

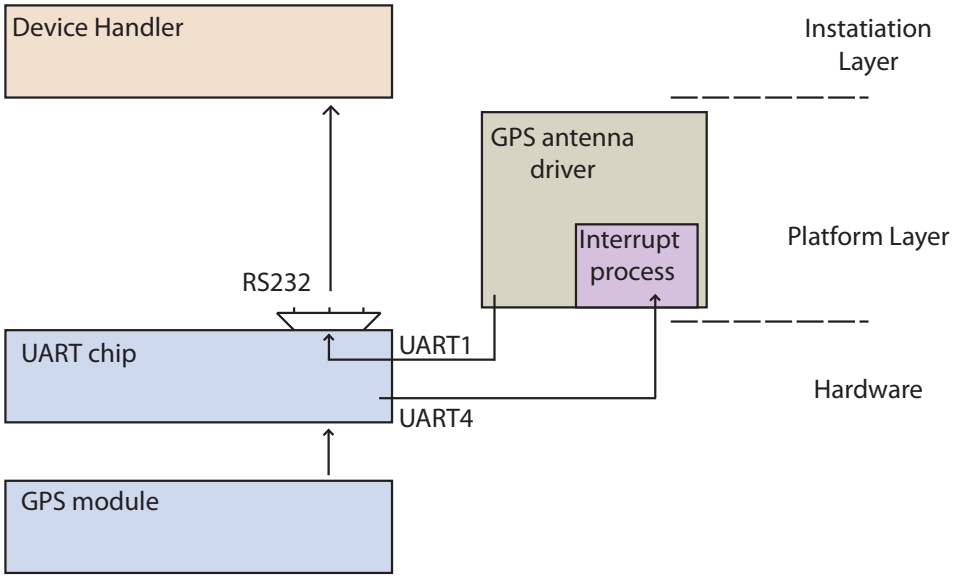

Figure 3.8: The GPS module sends data to the driver via UART4 of the UART chip. The driver returns the data to UART1 of the UART chip after an analysis. The data is sent via RS232 to the Device Handler.

character device[1], which makes it possible to mount it in a file system and use it as if it was a regular file. This allows making common function calls like open(), close(), read() and write() to the device. The rest is already done since there is a working device driver that handles the external UART chip. This solution still requires that the operating system is configured according to the bootloaders (see 2.4.2) settings and that "serdd" is connected to this driver. This character device, that hold data from the NMEA sentence, can be sent to the COM Handler.

The other way to manage the communication is to talk directly to the UART (according to 2.1.1) through OSE standard interface (device.h) using ddStart(), ddWrite() etc. This solution skips the serdd and the mounting to a file system layers. Instead you need to create a process context of the interrupt routine that is specified for the external UART chip. It is only necessary to create the context; you need not to specify what the process is doing. You make a BIOS call (again see 2.1.1) where you create a handle, specify the unit, create a pointer to a function etc. For every interrupt process you create a separate OSE interrupt process. You make a BIOS call to configure the unit and finally you make a BIOS call to start the unit. The function that was pointed to in the first BIOS call will be called every

[1]A peripheral device that transfers data one byte at a time, such as a parallel or serial port.

time the GPS receiver sends data to the UART port. So the final step is to write a make this function forward (or analyze the data and then send) the data to the COM Handler.

The device driver could be able to receive both NMEA messages and UBX messages from the GPS module. This is a decision to make, since the rest of the system is uninterested in UBX messages. If no UBX messages will be passed on to the rest of the system, the driver can simply ignore these data or throw it away. The default settings configure the GPS module to be able to transmit both NMEA messages and UBX messages, but no UBX messages are activated at start up. These must be activated via UBX input messages or with GPSMODE pins (the 7 configuration pins determining the start up settings). To build the minimum platform, no UBX messages will be received by the device driver from the interrupt process.

However the device driver should be able to send UBX messages to the GPS module to make configuration changes to the GPS receiver hardware. It includes settings related to the configuration pins used to determine the GPS module's boot settings. These are settings of for example update rate, baud rate, TIME PULSE and sentences used.

The analysis that can be made of the received data is first and foremost to determine what kind of message that has been received. How this analysis can be made is covered by the text describing how to write a message parser (see section 2.4.2). The part of the message that differs is the three-letter identifier. Since only a few sentences are used in the system, these are the only ones that will be sent to the Device Handler. All the other sentences that are received by the device driver will be thrown away.

If the message is to be analyzed a memory allocation of a capacity of at least a full sentence is created. The incoming data is stored with start at the first position to be filled continuously until a "new line" character is collected. Then you know that the entire sentence has been collected and the data can be passed on to the system if it was complacency. If a dollar sign is collected that means you have got the first character of a new sentence and you should store that on the first position after you have thrown the old data away.

Chapter 4

Result

For the i.MX31ADS hardware there is a driver implemented for the graphics port to Linux that support displays with the same control signals (VSYNC, HSYNC) as the SHARP screen that was used during this work. The framebuffer driver, which activates the LCD Controller driver, is used by most graphics libraries (Qt, C/PEG, etc.) and let applications talk directly with the systems framebuffer. The already implemented touch screen driver also works with the panel, and other displays, which use the four pins on the LCD connection port that SHARP uses.

The long term goal of my thesis was that in the future there will be an opportunity to introduce graphics in the framework SHAPE. The short term objective was, theoretically and practically, to investigate the possibilities of achieving this goal. That an implementation was made in Linux for the hardware is also a way to show that it would work, because the graphics libraries are not operating system dependent, and they communicate with the system's framebuffer via a framebuffer driver. The framebuffer works in the same way in Linux and OSE, so a working driver to OSE would have given the same outcome. Unfortunately there is no working driver for the graphics port to ENEAs operating system OSE for the hardware i.MX31ADS.

The system works after the CAN protocol header change. However, no tests have been done to evaluate if the system has become more sustainable and lasting.

The work with the GPS antenna has had a focus on theory and design choices. A driver for the built-in GPS antenna on the Freescale hardware with an i.MXS processor was not implemented because of problems with the hardware.

4.1 Discussion and Conclusions

4.1.1 Graphics

Generally, the display is connected to the to the hardware's integrated LCD Controller (in the processor), with a flat panel cable, that pick data from the framebuffer (RAM). The LCD Controller driver is the software that initiates the LCD controller (the hardware register values, addresses and size of the framebuffer (LCD specific information)) and then do not need do anything more. Both for Linux and OSE, there is a system framebuffer.

For OSE, there is a framebuffer driver to the hardware i.MX21ADS, which manages the allocation of RAM for the framebuffer, and also initiate the LCD controller and then inform interested clients (e.g. graphic library for example) where the framebuffer is located. This driver use the signal based interface (fbdev.sig) described in section 2.1.1. The hardware's (i.MX21ADS and i.MX31ADS) has many similarities but also some differences. Still, the work on the i.MX21ADS is a basis to achieve the corresponding functionalities for the i.MX31ADS.

Restrictions

The work is based on an existing BSP for the Linux operating system. The BSP had initially lots of functionality that was related to graphics and the framebuffer.

Graphics support for the operating system OSE exist in a small extent within the company. The implementations that are, of course, is to other hardware and none of these implementations make use of the IPU.

The i.MX31 includes a Graphic Processing Unit (GPU), which supplement the processor by hardware accelerating 2D and 3D graphics. In order to take advantage of what OpenGL ES has to offer it is required to use the built-in graphics accelerator on the application development system. For the used BSP for Linux the drivers for the graphics accelerator were delivered as a binary (already compiled code). Along with the graphics port driver now available for Linux, it is possible to write OpenGL ES applications.

The source code was needed to be able to implement drivers for the graphics accelerator in OSE and further to be able to be able to run OpenGL ES applications with satisfactory results in OSE. To simply use the CPU will cause the system not cope with the calculations required and other parts of the system would suffer.

Improvement opportunities

Because of the screen properties with need of continuous refresh signals from the LCD controller, the screen consumes about as much power when

the same image appear a long time as if the screen is constantly updated with information from the framebuffer.

Although this is a problem with the panel, it can be balanced by features that save energy in other ways. The ability to turn off the screen in a simple way should be available and tasks that are able to be moved from the processor to any other part of the system should be implemented, since graphics is demanding on the processor. It would be a large improvement of the system as a whole if all the graphics used the built-in graphics accelerator.

Use in other systems

The system that is now functioning has wide possible use. Most graphics libraries rely on a framebuffer which applications can write to, possibly via hardware accelerated components, to present graphics. The framebuffer is quite generally implemented, but some variables must of course be tailored to the hardware, like the size of the framebuffer in memory, etc. However, the LCD controller driver is very adapted to the hardware and this driver is what differentiates most between two hardwares.

Conclusions

There are many graphics libraries to choose from on the market. Based on the requirements set out in report

SWRQ 1 - It shall be suited for embedded systems

SWRQ 2 - Operating system independent

SWRQ 3 - It should be able to create both 2D and 3D graphics

SWRQ 4 - It shall be an open standard

there are basically two graphics libraries that meet these requirements, namely OpenGL ES and Mobile 3D Graphics API for Java. The recommended choice to make is to use OpenGL ES, as it is a low level graphics library and does not have the same overhead compared to Mobile 3D Graphics API for Java. Mobile 3D Graphics API for Java is not a low-level graphics library and is not recommended to use on its own. However it can be combined with OpenGL ES since both graphics libraries can take advantage of the same underlying rendering engine. This would increase the system's use, since there is a large market for Java games and similar in other areas, such as mobile telephony etc. That is the answer to question "*Q1. What graphics library is appropriate to use in SHAPE?*" that was posted in the introductory chapter of the report.

The question "*Q2. What choices can be done to reduce needed system resources for graphics?*" has been answered with a non-software solution.

Embedded software is very dependent on the hardware of a product. By using dedicated hardware acceleration for floating point calculations and dedicated graphics hardware you can get faster execution and lower power consumption. When using e.g. OpeGL ES, there is EGL that allows interfacing and control of resources such as framebuffers, memory, window and platform resources. Reducing the unnecessary use of off-screen drawing surfaces contribute to better use of system resources.

Implementation-wise, a driver for the graphics port for Linux is implemented. A corresponding driver for the graphics port was not done to OSE. This has however not been a problem, since the operating systems work the same way when it comes to graphics. Graphic libraries write data to the systems framebuffer and this is sent to the display.

4.1.2 CAN

System appearance

The protocol that has been analyzed and changed was already in use in an existing system. The system is not large and has at some points not taken into account (slightly simplified) factors which may occur in larger systems with more diverse hardware and resources. The system makes use of hardware with the same limitations and requirements on the CAN protocol header. By that I mean, for example, that no difference exists in the endian. The number of nodes and processes are so far limited. Would a larger system be used, more information for both nodes and processors ought to be allowed in the CAN protocol header. Now is that information space rather unused relative to the space allowed. For future use, within the project framework, a more robust network communication channel is required.

Improvements

A balance must always be made to prioritize what is considered to be a priority. If the system changes or if other software is changing conditions, such as it can take over parts of priority or similar, then there are improvements to be made. As mentioned, the system is composed of units that have the same hardware requirements and thus does not use all bits of the protocol (endian). It is also not as many nodes in the system as permitted by the protocol, but it is a future solution.

Benefit in other situations

The system uses only the standard CAN message format, hence no extended format. That is a limitation that also reflects on the change that has been made in the CAN protocol header. The CAN protocol header is also designed solely based on the use of the system today. The hardware register

that the design has been implemented on is specific to the existing hardware. However, both the design and implementation is easy to transfer to a different hardware with similar conditions.

Conclusions

The theoretical work in the field of CAN communication has been driven by the question "*Q3. What limitations/shortcomings imply communication through CAN in DySCAS?*" Two main problems were identified in the existing CAN communication. The signal IDs was not assigned in a predefined manner and the priority bit had the greatest impact of the message priority. This was corrected by identifying all the signals and then assigns them new signal numbers based on how important and how often the signals are used. The CAN protocol header is also redesigned so that group, system signals and application signals, have the greatest impact on the signal priority.

4.1.3 GPS

Time was put on getting ENEAs OS to boot on the hardware. The operating system should have functioned in the past, but attempts yielded no positive results despite the help of experienced workers. There are several possible explanations to why the operating system did not start, ranging from faulty hardware to faulty software or configuration of the system. The error occurred was that reading and writing of the memory (within the configured area of RAM) resulted in that errors were thrown and the system would not start. No new software was used when attempting to get the operating system to start.

Conditions

The hardware belonged to a third party company and the time with the hardware was limited. The time with the hardware was not enough and the decision was taken to leave the task and instead spend time on other tasks in the thesis framework. The estimated time (in my time plan) to be used to implement the driver was used for troubleshooting.

Conclusions

An area extending beyond the boundaries of this master thesis is the answer to the question "*Q4. What features can a GPS contribute to DySCAS?*" The most obvious use of a GPS receiver is to exploit the time information and to use the information about the location, combined with map software, to give driving directions. In DySCAS a great way to extend the areas of use of a GPS receiver is to combine the technology with other technologies (e.g. cell phones). A possible feature is to plan your trip at home by using

another GPS receiver and to transfer that information when you arrive to the vehicle. This could also be made in the opposite direction, if you are heading somewhere by foot after having traveled a distance by car. Information from the GPS receiver can be used to predict how long it will take before the vehicle reaches a certain point. The scope of use is more or less endless depending on the number of devices you can combine the GPS with.

The GPS used in the thesis can use the NMEA protocol for output to the system. It will fit to the existing COM port handler that receive signals from a serial cable and after a complete message is sent, this will be sent forward to an application that analyzes the string from which the information requested. One part of the analysis is covered by the question "*Q5. How do you pick out data from the GPS stream?*" The NMEA specification allows several appearances of the data fields, but they are always separated by decimal points. The easiest way to extract information from the data stream is to write a parser that reads out the information between two commas. This way you can have as many digits in an individual data field as you want or even don´t have any information at all in a data field. To change settings in the GPS hardware communication is made through the same serial port, but with the UBX protocol.

4.2 Future Work

4.2.1 CAN

To make the system more compatible with other hardware, a verification of the hardware endian in the CAN communication could be implemented. In the current situation this bit is not used in the CAN protocol header, but has been saved for future use. Either such a check should be made (and the bit set, which is not done today) or that bit can simply be used for any other fields in the protocol. The endian bits role in the header is the order in which the subsequent data frame is to read.

4.2.2 Graphics

A corresponding graphics port driver could be created to the operating system OSE by porting the Linux code unit for unit (it is possible to make tests for certain parts separately) or getting the graphics to work together in OSE and then move parts into the framework of SHAPE.

The i.MX31ADS hardware also supports the use of multiple displays simultaneously. This support can be implemented in software terms and utilized in the future.

Within ENEA there are at least four application development systems with partial support for graphics (Omap5912_osk, i.MX31ADS, i.MX21ADS, Redhawk). The hardware with most similarities to i.MX31ADS is i.MX21ADS,

which actually has support for the same display as the one used in this work. There is within the company a written touch screen driver to the i.MX21ADS hardware (perhaps the same display as used in this work), which corresponds to the Linux driver for the i.MX31ADS. It can be moved over to i.MX31ADS when the screen is operational in OSE.

4.2.3 GPS

If the operating system will start on the hardware persist to implement the GPS anthenna driver. Based on the design proposal that has been given, the driver for the GPS antenna could be implemented. It should be linked with an existing GPS handler located in SHAPE middleware as an intermediary between an application and the driver. The GPS can use the protocols that are, or even expand the number of allowed varieties of NMEA sentences.

Bibliography

[1] Open source code/GLX. http://www.sgi.com/products/software/opensource/glx/.

[2] OpenGL Shading Language. http://www.opengl.org/documentation/glsl/.

[3] SungHo Ahn, DongMyung Sul, SeungHan Choi, and KyungHee Lee. Implementation of lightweight graphic library builder for embedded system. *Advanced Communication Technology, 2006. ICACT 2006. The 8th International Conference*, 1:3 pp.–, Feb. 2006.

[4] T. Capin, K. Pulli, and T. Akenine-Moller. The state of the art in mobile graphics research. *Computer Graphics and Applications, IEEE*, 28(4):74–84, July-Aug. 2008.

[5] Fadi Chehimi, Paul Coulton, and Reuben Edwards. Evolution of 3D mobile games development. *Personal Ubiquitous Comput.*, 12(1):19–25, 2008.

[6] ENEA Embedded Technology. *C166 Board Support Package User's Manual*.

[7] ENEA Software AB. *OSE Core User's Guide*.

[8] ENEA Software AB. *OSE Device Drivers User's Guide*.

[9] ESD-Electronics. Controller area network, A serial bus system - not just for vehicles. www.esd-electronics.com/german/PDF-file/CAN/Englisch/intro-e.pdf.

[10] Fabio Estevam. *Using OpenGL Applications on the i.MX31ADS Board*. Freescale, April 2008.

[11] Freescale Semiconductors. *Data Sheet: Technical Data MC9328MXS*, Rev. 3 edition, December 2006.

[12] Freescale Semiconductors. *i.MX31ADS Application Development System, User's Manual*, Rev. 1(a) edition, March 2006.

[13] JSR 184 Expert Group. Mobile 3D Graphics API for J2ME. http://www.jcp.org/en/jsr/detail?id=184.

[14] The Khronos Group. Open Standards for Media Authoring and Acceleration. http://www.khronos.org/.

[15] Infineon Technologies. *C167CR Derivatives 16-Bit Single-Chip Microcontroller*, v. 3.2 edition, May 2003.

[16] S. Vassiliadis L. Antochi, B. Juurlink and P. Liuha. GraalBench: A 3D graphics benchmark suite for mobile phones. In *LCTES'04*, 2004.

[17] Sing Li and Jonathan Knudsen. *Beginning J2ME^{TM}: From Novice to Professional, Third Edition*. APress, Berkeley, 2005.

[18] R. Moller. State-of-the-Art 3D Graphics for Embedded Systems. *Devices, Circuits and Systems, Proceedings of the 6th International Caribbean Conference on*, pages 339–343, April 2006.

[19] National Instruments. *NI-CAN®Hardware and Software Manual*, October 2006.

[20] Antti Nurminen. m-LOMA - a mobile 3D city map. In *Web3D '06: Proceedings of the eleventh international conference on 3D web technology*, pages 7–18, New York, NY, USA, 2006. ACM.

[21] K. Pulli, T. Aarnio, K. Roimela, and J. Vaarala. Designing graphics programming interfaces for mobile devices. *Computer Graphics and Applications, IEEE*, 25(6):66–75, Nov.-Dec. 2005.

[22] Kari Pulli. New APIs for mobile graphics. In *Proc. SPIE Electronic Imaging: Multimedia on Mobile Devices II,SPIE*, pages 1–13, 2006.

[23] Kari Pulli, Tomi Aarnio, Ville Miettinen, Kimmo Roimela, and Jani Vaarala. *Mobile 3D Graphics with OpenGL ES and M3G*. Morgan Kaufmann, 2007.

[24] GPS service daemon. a list of NMEA sentences. http://gpsd.berlios.de/NMEA.txt.

[25] M. Snyder. Solving the embedded OpenGL puzzle - making standards, tools, and APIs work together in highly embedded and safety critical environments. *Digital Avionics Systems Conference, 2005. DASC 2005. The 24th*, 2:9 pp. Vol. 2–, Oct.-3 Nov. 2005.

[26] Cheng-Han Tu and Bing-Yu Chen. The architecture of a J2ME-based OpenGL ES 3D library. *Computer Aided Design and Computer Graphics, 2005. Ninth International Conference on*, pages 5 pp.–, Dec. 2005.

[27] u-blox. *TIM-LA GPS Receiver Module, Data Sheet.*

[28] Monte Variakojis. Nmea parser design. www.visualgps.net/Papers/NMEAParser/NMEA%20Parser%20Design.pdf.

Appendix A

Abbreviations

ADS - Application Development System

API - Application Programming Interface

ASCII - American Standard Code for Information Interchange

BSP - Board Support Package

CSMA/CD - Carrier Sense Multiple Access with Collision Detection

CSMA/CR - Carrier Sense Multiple Access with Collision Resolution

CAD - Computer-Aided Design

CAN - Controller Area Network

CPU - Central Processing Unit

DySCAS - Dynamically Self-Configuring Automotive System

EGL - Embedded-System Graphics Library

J2ME - Java 2 Platform Micro Edition

GIS - Geographical Information System

GLX - OpenGL Extension to the X Window System

GNSS - Global Navigation Satellite System

GPS - Global Positioning System

GPU - Graphic Processing Unit

HMI - Human-Machine Interface

IPU - Image Processing Unit

LAR - Lower Arbitration Register

LCD - Liquid Crystal Display

LCDC - Liquid Crystal Display Controller

NMEA - National Marine Electronics Association

PDA - Personal Digital Assistants

SHAPE - Self-configurable High Availability and Policy base platform for Embedded systems

UAR - Upper Arbitration Register

VGA - Video Graphics Array

WGS84 - World Geodetic System 1984